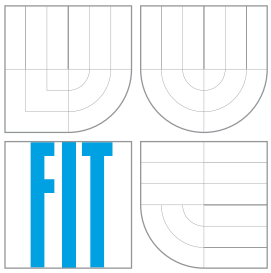
BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ



FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

RECONSTRUCTION OF REPETITIVE DNA SEGMENTS

REKONSTRUKCE OPAKUJÍCÍCH SE SEGMENTŮ DNA

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR
AUTOR PRÁCE
Bc. ROBERT BIKÁR

SUPERVISOR
VEDOUCÍ PRÁCE
Ing. TOMÁŠ MARTÍNEK, Ph.D.

BRNO 2016

Master Thesis Specification/18874/2015/xbikar00

Brno University of Technology - Faculty of Information Technology

Department of Computer Systems Academic year 2015/2016

Master Thesis Specification

For: **Bikár Robert, Bc.**
Branch of study: Bioinformatics and biocomputing
Title: **Reconstruction of Repetitive DNA Segments**
Category: Biocomputing

Instructions for project work:

1. Get acquainted with basic principles of molecular biology and mechanisms of storing information in DNA sequences.
2. Study related works in the area of reconstruction of repetitive DNA segments.
3. Modify existing algorithm or develop a new algorithm for reconstruction of repetitive DNA segments.
4. Implement the proposed algorithm in a suitable programming language
5. Verify the functionality of implemented tool on real data sets.
6. Finally, discuss properties of created tool and possibilities for future work.

Basic references:

- According to instructions of the supervisor.

Requirements for the semestral defense:

- Items 1 to 3.

Detailed formal specifications can be found at http://www.fit.vutbr.cz/info/szz/

The Master Thesis must define its purpose, describe a current state of the art, introduce the theoretical and technical background relevant to the problems solved, and specify what parts have been used from earlier projects or have been taken over from other sources.

Each student will hand-in printed as well as electronic versions of the technical report, an electronic version of the complete program documentation, program source files, and a functional hardware prototype sample if desired. The information in electronic form will be stored on a standard non-rewritable medium (CD-R, DVD-R, etc.) in formats common at the FIT. In order to allow regular handling, the medium will be securely attached to the printed report.

Supervisor: **Martínek Tomáš, Ing., Ph.D.**, DCSY FIT BUT
Beginning of work: November 1, 2015
Date of delivery: May 25, 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2

L.S.

Zdeněk Kotásek
Associate Professor and Head of Department

Abstract

The main motivation for master's thesis is to find suitable algorithm that creates a graph representation of NGS sequencing data in linear time. De Bruijn graph was chosen as a method for research. Next, the tool was designed to be able to transform the graph and correct errors created during construction of the graph. The main aim of the thesis is to implement a tool that reconstructs repetitive segments in DNA. Implemented tool was tested and is able to identify repetitive segments, specify types, visualize them properly and is also able to assemble their sequence with fine accuracy on simpler genomes. When using complex genomes, tool is able to reconstruct only fragments of repetitive segments.

Abstrakt

Hlavní motivací diplomové práce bylo najít vhodný algoritmus, který by vytvořil grafovou reprezentaci NGS sekvenačních dat v lineárním čase. Zvolenou metodou pro reprezentaci je de Bruijnův graf. V další části práce byl navrhnut nástroj, který je schopen transformovat graf do přijatelné podoby pro vykreslování, a dále je schopen odstraňovat chyby, které vznikají při konstrukci grafu. Cílem práce je vytvořit nástroj, který rekonstruuje repetitivní segmenty v DNA. Implementovaný nástroj byl otestován a je schopen identifikovat opakující se segmenty, určit jejich typy, vizualizovat je a sestavit jejich sekvenci na jednodušších genomech s velkou přesností. Při použití složitějších genomů, nástroj nalezne pouze fragmenty repetitivních segmentů.

Keywords

bioinformatics, DNA, repetitive elements of DNA, transposon, DNA satellite, de Bruijn graph

Klíčová slova

bioinformatika, DNA, opakující se elementy DNA, transposon, DNA satelit, de Bruijnův graf

Reference

BIKÁR, Robert. *Reconstruction of Repetitive DNA Segments.* Brno, 2016. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Martínek Tomáš.

Reconstruction of Repetitive DNA Segments

Declaration

Hereby I declare that this masters's thesis was prepared as an original author's work under the supervision of Mr. Tomáš Martínek. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Robert Bikár
May 23, 2016

Acknowledgements

I would like to thank Mr. Tomáš Martínek for his help during solving the thesis, for his valuable guidance and hints and time spent on consultations.

Contents

Introduction

Repetitive DNA segments are patterns in DNA that are spread throughout the genome in multiple copies. These segments are very frequent in eukaryotic and prokaryotic organisms and occupy significant part of the genome. Detection of these sequences could be crucial for full understanding of the genomic information structure and its complete functionality. Reconstruction of the repetitive segments is a challenging computational task.

Next generation sequencing methods must handle these segments very carefully and effectively in order to assemble genome correctly. Several types of methods have been used so far. All to all sequences similarity and clustering approaches are commonly used. RepeatExplorer [19] and Transposome [22] use that approach but time and space complexity of algorithms used are too high. Therefore, they cannot be efficiently used for very large genome with extreme diversity and high number of repeats. Another tool Tedna [26] uses de Bruijn graph principles for sequences representation and repetitive segment reconstruction.

The aim of this thesis is to perform a research of existing methods for repetitive DNA segments reconstruction, design, implement algorithm for reconstruction. Method proposed in the thesis is based on de Bruijn graphs. This approach is memory efficient and does not require extreme computational resources. De Bruijn graph and k-mers are used for design and implementation of reconstruction of repetitive DNA segments. The results of reconstruction are very accurate for simpler sequences, reconstruction of repeats in more diversificated genome founds only fragments of repeats.

In chapter 1 a brief summary of repetitive DNA types is provided. It describes particular classes and types of repeats. Chapter 2 summarizes existing tools for repetitive segments reconstruction and proposes thesis direction. Following chapter shows a theoretical background of de Bruijn graph itself, its construction process and properties (see chapter 3). There are also definitions of algorithms used for graph transformations and description of possible errors created during de Bruijn graph creation and their possibility of elimination. In chapter 4 there is a proposal of design and architecture of the program. Each component is specified with its requirements and functions. Chapter 5 describes the implementation of each component - method of input reading, algorithms and data structures for de Bruijn graph creation, transformations and visualization, method for repeat reconstruction. In the chapter 6 implementation of generating testing genome is described. It is followed by evaluation of graph transformations and its influence for visualization. In the last part of chapter 6 repetitive reconstruction results are shown and evaluated. There are various test cases that differ in used genome, repeats and mutations.

Chapter 1

Repetitive DNA segments

Eukaryotic genome contains an abundance of repeated sequences. They are divided into two groups:

- **Transposable elements** also called transposons (TEs) are able to move from one place in genome to another one. Transposons have been identified in eukaryotic organism as well as in prokaryotic organisms. Occupancy of transposons in the genome can be very high, for example transposable elements occupy 10% genome of some fish species, 37% of the mouse genome, 45% of the human genome and up to 80% of the plants genome like maize [18].
- **Satellites** – tandemly repeated segments that do not move in the genome. They create very large arrays and are the main component of functional centromere. Length of the base repeat vary from 1bp to thousand of base pairs, repeated 5-100 times. According to the length, satellites can be further divided into microsatellites (base segment length: 1-5bp) and minisatellites (base segment length: 5-50bp) [12].

Transposons are capable to make significant changes in organism's genome because of their movements along the genome. During the phase of transposition and incorporation back to genome several mutations occur like insertion, deletion or duplication [15]. Transposable elements of eukaryotic organisms can be divided into two major classes accordingly how they are transposed. Class I elements are transposed by RNA - these elements are designated as retrotransposons or retroelements. Class II elements are transposed by DNA and they are designated as DNA transposons. Both classes of transposons contain autonomous and non-autonomous elements. Autonomous elements consist of ORFs (Open reading frame) in which required products for transposition are coded. On the other hand, transposition proteins are not coded in non-autonomous elements. However, non-autonomous are able to transpose because of cis sequences possession[1] [23].

1.1 Class I repeats

Class I transposons use RNA for transposition. Typically, transposition method is replicative and is called "copy-and-paste". Hence, retrotransposons are copied in genome and create many repeated sequences. Diagram of transposition cycle is described on figure 1.1. Retrotransposons can be divided into two subclasses according to presence of long terminal repeats (LTRs) which are at the beginning and at the end of retrotransposon body [20]:

[1] cis sequence is required for transposition

- **LTR transposon** – body of transposon is surrounded by LTRs.
- **Non-LTR transposons** – body of retrotransposon is not surrounded by LTRs.

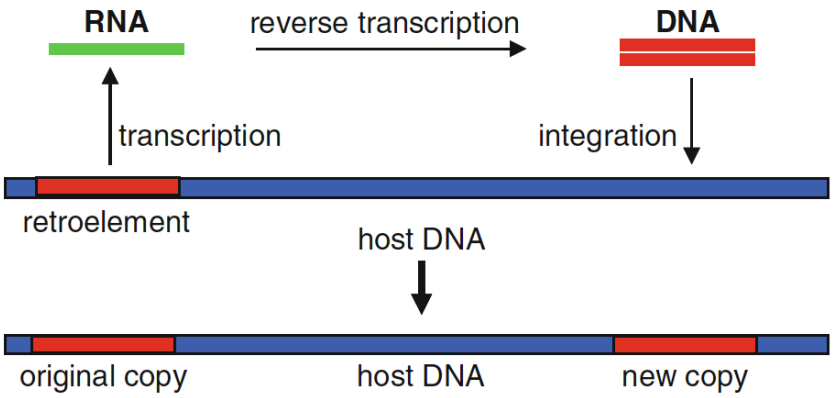


Figure 1.1: „copy-and-paste“ trasposition mechanism of LTR transposons [14].

LTR Transposons

Transposons of this class have been identified in plants. LTR transposons can be divided into two major families *Ty1–gypsy-like* and *Ty1-copia-like*. Both types contain the same protein coding domains only their order is different. Both families of transposons code *gag* and *pol* genes. Difference of order was detected in *pol* gene which codes several enzymes. Structure is shown on figure 1.2. A list of coded enzymes follows:

- PR – protease,
- RT – reverse transcriptase,
- RH – ribonuclease H,
- INT – integrase.

These enzymes manage the process of reverse transcription and integration of daughter sequence to new place in chromosome. Size of LTR transposons range from hundreds of base pairs to as much as 25 kbp. Surrounding LTRs have length range in hundreds of base pairs to 5 kbp.

Retroviruses and LTR transposons share many features – comparison of LTR transposon structure (gypsy and copia family) with retroviruses – see figure 1.2. The major difference between retroviruses and LTR transposons is that LTR transposons do not contain Envelope protein (ENV) [14].

Non-LTR Transposons

Class of non-LTR transposons also called as retroposons consist of two sub types – long interspersed nuclear elements (LINEs) and short interspersed nuclear elements. Both sub

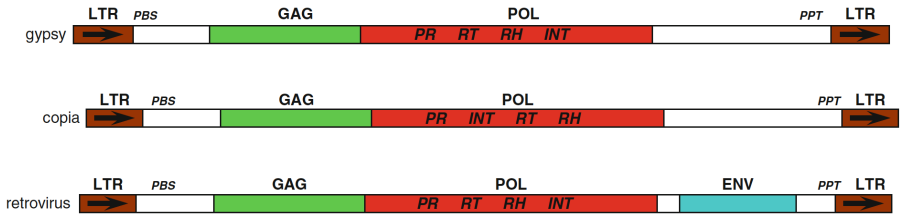


Figure 1.2: Structure of LTR transposons compared to retrovirus [14].

types can differ in structure as seen on figure 1.1. Firstly, non-LTR transposons were discovered in mammalian genomes but also have been identified in plants, fungi and invertebrates [21].

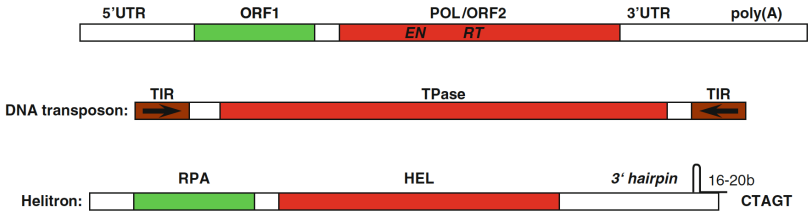


Figure 1.3: Structure of non-LTR transposon compared to DNA transposon and Helitron [14].

LINEs

Long intersperses nuclear elements, also designated as LINE-like sequence or L1, have no LTR but contain poly(A) tail – definition of 3' terminus. LINEs L1 family are highly repetitive elements that affects mammalian genomes. Families R1 and R2 are widely spread in insects and retroelements F, I and Jockey of *Drosophila melanogaster*. The length of LINEs is about several kilobases and every LINE contains two ORFs. ORF1 is composed with gag protein, ORF2 contains endonuclease and reverse transcriptase domains. These ORFs combined are the key of autonomous retro transposition of LINEs [21].

SINEs

Next type of non-LTR transposons are short interspersed nuclear elements. Their length is up to several hundreds of base pairs. The structure of 5' region is similar to tRNA genes. As shown on animals that region is similar to 7SL RNA genes. 3' region in many SINEs is very similar to the 3' end of LINEs. The termination region of SINEs consists of poly(A) tail or sequences rich on A or T nucleotides. There is no coding sequence for reverse transcription in SINEs so they are not able to transpose on their own. SINEs are dependent on activity of reverse transcriptase included *in trans* [24].

1.2 Class II repeats

Class II transposons use DNA for transposition. The mechanism used for transposition is called "cut-and-paste". Transposon is literally cut from its original position and then

integrated to the new location. The method is shown on figure 1.4.

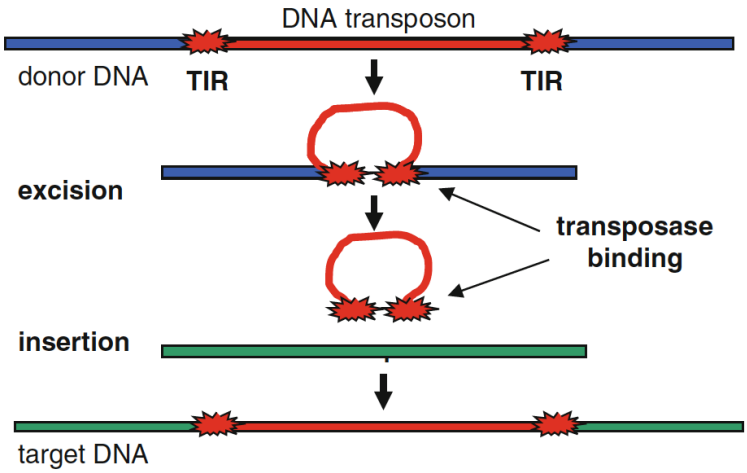


Figure 1.4: „cut-and-paste“ transposition mechanism [14].

Structure of Class II transposons is pretty simple. Element consist of transposase gene surrounded by two TIRs (terminal inverted repeats). These TIRs are recognized by transposase and then DNA body of transposon is excised and inserted to the new location. After insertion, target site of DNA is duplicated which results in Target Site Duplications (TSDs). They represent unique marker for each DNA transposon. DNA transposons are classified into two sub classes depending on their sequence, TIRs and TSDs. Subclass I contains following families of DNA transposons: Tc1/mariner, PIF/Harbinger, hAT, Mutator, Merlin, Transib, P, piggyBac and CACTA. Subclass II consist of Helitron and Maverick transposons because they use different mechanism of transposition. Some significant families of DNA transposon are described below [18].

Subclass I

Superfamily Tc1/mariner

Elements of Tc1/mariner family are very widely spread in all organisms. They are from 1 to 5 kb long. Every element is flanked by two TIRs (17 to 1100 bp). Sequence of transposase proteins differ between Tc1/mariner family elements but all of them contains two characteristic domains: an amino-terminal region with the helix-turn-helix (HTH) motif which is required for TIRs recognition and binding. Second domain is a carboxy-terminal containing a catalytic motif made of three amino acids - in the case of mariner type elements, or DDE in the case of Tc1 element type.

piggyBac

DNA transposon piggyBac have been identified in genome of *Trichoplusia ni*. Transposition mechanism, structure and control is similar to the Tc1/Mariner family elements. piggyBac elements are 2.4 kbp long and TIRs are 13 bp long. There are also additional internal inverted repeats (19 bp). In these transposons functional transposase is located in single ORF element (1.8 kb).

hAT

DNA transposon hAT (hobo/Ac/Tam3) have been found at eukaryotes. They are 2.5 to 5 kb long and encode a transposase containing a catalytic DDE motif and DNA binding domain BED zinc finger. The transposase gene is surrounded by TIRs 5 to 27 bp long. TSDs have also heterogenic sequences (8 bp) [20].

Subclass II

Helitrons

Elements of Helitrons families do not generate target site duplication as other eukaryotic transposons do but helitrons target AT sequence. This method does not lead to any duplication of this sequence. The mechanism of transposition is called rolling circle which affect only one strand of DNA. This principle is shown on figure 1.5. Helitrons are very difficult to be recognized because of a lack of a typical structure of DNA transposons. On the 5' end there is TC motif and on the 3' end there is CTRR motif (R stands for purine) [18].

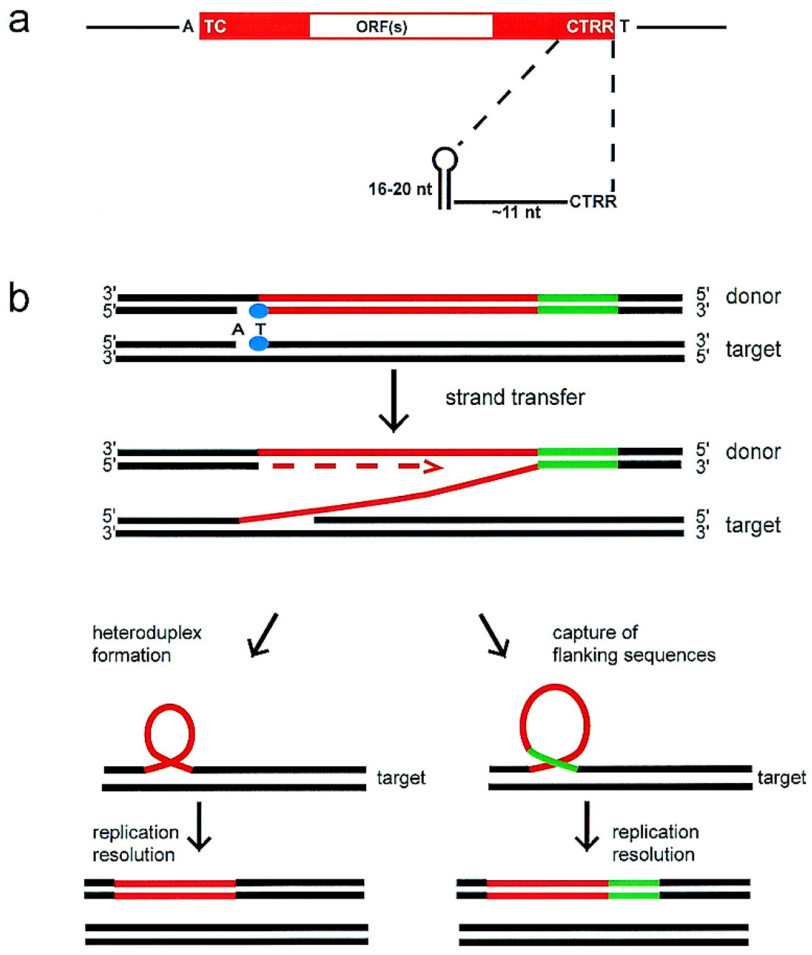


Figure 1.5: Rolling circle mechanism of Helitrons transposition [11].

Chapter 2

Tools for repetitive segments reconstruction

In this chapter, existing tools for repetitive segments reconstruction are described. Because of the aim of the thesis especially *de novo* assemblers of transposable elements are highlighted: RepeatExplorer, Transposome and Tedna. Principle of each tool is provided and is followed by evaluation of results and comparison of advantages and disadvantages. At the end of the chapter, described tools are summarized and proposal of thesis direction is provided.

2.1 Tools overview

Each tool uses mostly pair ended reads as input which are created by Illumina dye sequencing technique [7]. Described tools do not require any reference database of known repetitive segments.

RepeatExplorer

Repeat Explorer is a set of software tools for identification and characterization of repetitive segments. It is a web application running on online platform Galaxy that provides user interface for computational methods for genome analysis. Via web interface user is able to execute analysis, create documentation and share results. Algorithms use graph based clustering methods. Clustering is performed using Louvain method [10]. As input RepeatExplorer uses short reads that are chopped from genome. So the NGS data are very suitable for this tool. Reads are either single or pair ended with uniform length [19]. Process of repetitive segments identification comprise several steps:

- First, all to all similarity comparison is computed between individual reads.
- Graph-based clustering is performed in order to separate graph into clusters (groups) to identify particular communities in the graph. Each cluster represents a family of repeat.
- Graph topology is further analyzed according to proportion of genome and similarity to known repetitive elements and conserved protein domains. Clusters are compared by similarity against RepeatMasker and RepBase databases [13]. RPS Blast search is used to comparison against conserved domain database but this is very inefficient and slow method, Blastx search against database of protein domains is used instead.

- Graph layout is computed in order to create reasonable visualization which helps user to analyze graph by interactive exploration that is also very essential part of RepeatExplorer.
- Large repeats seem to be a problem because they are usually split into more clusters. Then re-clustering is performed with user aid, cluster similarities are computed and information about pair ended reads also helps with deeper analysis.
- Assembly of repetitive segment is done by CAP3 program.

Repeat Explorer is able to identify high and medium copy repeats in plant genomes, create repeats annotation and determine their quantification. Annotation can be created also automatically via neural networks. Very sophisticated and user friendly interface is a great advantage. On the other hand, used algorithms (all-to-all similarity computing and graph-based clustering) have extreme memory consumption and require much computational performance. Time and space complexity of methods is at quadratic level $\mathcal{O}(n^2)$. There is also limit of reads that could be processed in one run. This is dependent on genome composition, on number of repeats, copy number, length and diversity.

Transposome

Transposome is a command line application that is based on RepeatExplorer approach but provides more efficient and more accurate solution. As an input Transposme uses reads created by whole genome shotgun method – WGS. The coverage of input data is expected to be very low (1%). Analogous to RepeatExplorer method, Transposome computes all to all similarity between sequence by highly parallel method. A modified version of megablast algorithm is used – mgblast which is memory efficient. Then graph based clustering is executed. Used method is very efficient, it uses edge weight information and time and space complexity is linear. Although this method is very efficient, it over-refines cluster that are separated to more groups. Original algorithm was modified by using information about pair ended reads, therefore unions in the graph are found. Each cluster now represents repetitive segment which is compared with reference library [22].

Transposome shows very efficient methods for transposable element family identification from low coverage WGS data. Increasing data coverage results into better estimates. The very useful feature is that the results are immediately translated into very accurate approximation of genomic composition of repeats. Great advantage is that clustering algorithm complexity is linear – $\mathcal{O}(n)$. Annotation of transposable elements is also automatic. Also tool is able to analyze repeats directly from genome assembly. Transposome can be used as a general toolkit for NGS data and for creation of custom genome analysis pipeline. Transposome does not have any graphical user interface but provides Perl API for manipulation of input data, computing similarities and repeats annotation.

Tedna

Tedna uses different approach than tools described above. The approach is based on de Bruijn graph with analysis of highly repeated k-mers. As input Tedna uses pair ended reads and produce full length transposable elements. Repeats are reconstructed from de Bruijn graph where the most frequent k-mers are found. Tedna provides very good sensitivity and good specifity rate of found elements.

Tool is accurate for eukaryotes genome analysis with higher density of TEs. For now, Tedna only assemble repeated sequences but is not able to discriminate TEs, annotate of segments, discriminate genes of other repetitive elements. Tedna shows that de Bruijn graph approach is very efficient as we can avoid computing all to all similarity and has a great potential to future [26].

2.2 Existing tools summary and thesis direction proposal

Described tools show two possible ways how to reconstruct repetitive elements. There are tools that use all to all similarity principle and discriminate elements by clusters created with graph-based clustering. Although RepeatExplorer provide very good user interface and many auxiliary function, methods used are very inefficient and require too much computation power and memory. Next, Transposome uses the same principle but much more efficient algorithms were incorporated into pipeline. That solution might have some potential to the future.

Tedna proposed very different approach for repetitive segment reconstruction – the de Bruijn graph and k-mer frequency. This approach seems to be very efficient and has very good potential – it avoids all to all similarity computation, and the complexity of the de Bruijn graph creation is expected to be linear. Thus, because of the time and space efficiency and great potential this approach has been chosen for this thesis. De Bruijn graphs also creates challenging situations for graph transformations and non-trivial errors removal. Theoretical background that describes all aspects of de Bruijn graph is described in chapter 3.

Chapter 3

De Bruijn graph

This chapter theoretically defines de Bruijn graph and its construction. Possible errors that can occur during de Bruijn graph creation are described as well as means of error correction and removal. De Bruijn graph is named after Dutch mathematician Nicolaas Govert de Bruijn and is used in different fields science. De Bruijn graph principles are used to define grid network topologies, to implement distributed hash table or describe dynamical systems. But also de Bruijn graphs are used in bioinformatics - in *de novo* assembly of read sequences into a genome. In this thesis we use de Bruijn graph to represent a genome and reconstruct repeated sequences by their special properties.

3.1 De Bruijn graph theory

Definition 1. *Directed multi graph $G(V, E)$ consists of set of vertices V and multi set of directed edges E.*

De Bruijn graph is a directed multi graph thus we can denote node's in-degree as a count of incoming edges and node's out-degree as a count of outgoing edges. Node that has equal in-degree and out-degree is balanced. If in-degree and out-degree of node differ by 1 that node is semi-balanced [17].
Now we define basic building blocks of de Bruijn graph.

Definition 2. *K-mer is a substring of length k. Every k-mer represents one edge in the graph.*

Definition 3. *K-1-mer is a part of k-mer subtracted by 1-mer from left or right side. Left and right k-1-mers represent nodes in the graph.*

Example of combining k-mers and k-1-mers: Let $s = AGACTCG$ be an input string and $k = 3$. Then all k-mers of string s are: $s(k-mer) = (AGA, GAC, ACT, CTC, TCG)$. Every k-mer has its corresponding left and right k-1-mer. Constructed de Bruijn graph is shown on figure 3.1.

If input string is split into k-mers, we may lose information about order of some k-mer in the sequence. On the other hand, it is easier to order k-1-mers in final graph.

De Bruijn graph construction

Basic version algorithm that is used in thesis was taken over from [17] and works as described below:

- Input: list of sequences, k-mer parameter
- Output: de Bruijn graph represented by associative array
- Method:

```
for each read in the list of sequences:
  while the end of current read is not reached:
    chop read into left and right k-1-mer
    src := left k-1-mer
    dst := right k-1-mer
    if node(src) not exists:
      create node(src)

    if edge(src:dst) exists
      increase multiplicity(src:dst)
    else
      add new edge(src:dst)

    move to the next index of read
```

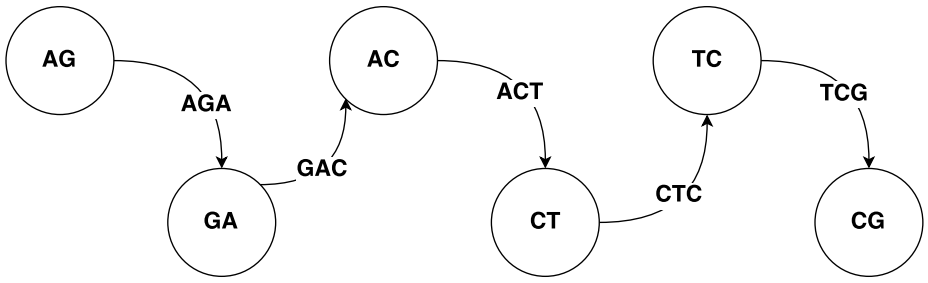


Figure 3.1: Simple de Bruijn graph.

Time complexity

Assume following process to compute time complexity of de Bruijn graph construction:

- For every k-mer create one edge between two nodes. We can assume that creation of an edge has constant time complexity - $\mathcal{O}(1)$.
- For every node and edge we get one hash - $\mathcal{O}(1)$.
- Key look up or key adding have also constant complexity - $\mathcal{O}(1)$.

Hence, total time complexity for one k-mer creation is constant $\mathcal{O}(1)$ and therefore in case of n k-mers complexity of whole graph creation is linear – $\mathcal{O}(n)$.

Space complexity

Space required for construction de Bruijn graph is $\mathcal{O}(min(N, G))$ where N is sum of lengths of all input reads and G is genome length. In case of high coverage G many time smaller than N. Therefore, space complexity is $\mathcal{O}(G)$.

Summary of graph construction complexity

- Construction of graphs is very simple; no special data preparation is needed, only input strings are needed (genome reads).
- Construction can be done in linear time - $\mathcal{O}(n)$.
- Total count of edges (k-mers) of graph is given by equation:

$$|E| = N - n * (k - 1)$$

  where N is sum of length of all reads, n is number of reads
- Maximal total count of nodes is given by another equation:

$$|V| = 2 * (N - n * (k\ - 1)) = 2 * |E|$$

  Total count of nodes is usually much lower because of repeated (duplicated) k-1-mers.

Construction problems

We may encounter several problems during de Bruijn graph construction.

Repeating nucleotide

If sequences of repeating nucleotides are found during graph construction, duplicate k-1-mers are created. Therefore, an edge connecting the node itself is created. This edge can be also multiple. Let $s = AAABBBBA$ be an input string and $k = 3$.

On figure 3.2 we can see that sequence AAA has identical right and left k-1-mer – so the cycle is created on vertex AA. Furthermore, for sequence BBBB which consists of 2 identical k-mers two cycles are created for vertex BB.

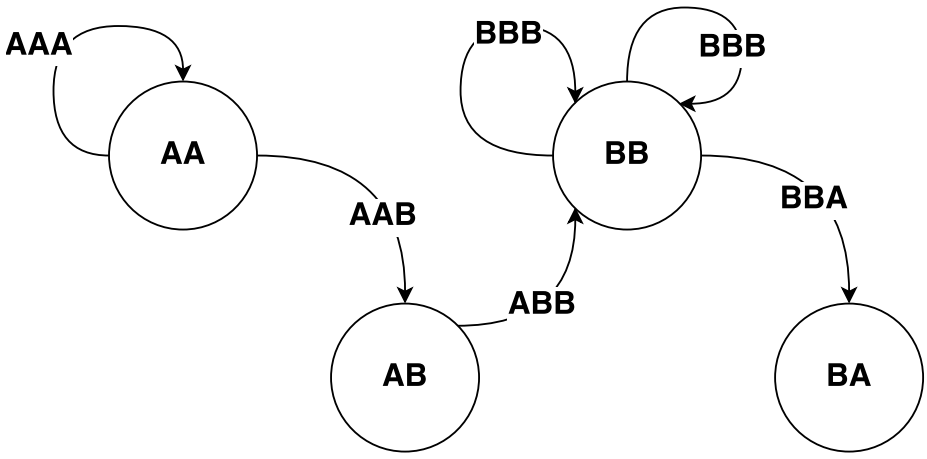


Figure 3.2: de Bruijn graph with repeating nucleotides in sequence.

Hence, we can assume following statement: If exists a repeating sequence of nucleotides of length N and this sequence is split into k-mers, the count of edges constructed from this sequence which create a cycle on vertex itself is equal to $N - k\ + 1$.

To get rid of duplicate edges a weight is added to each edge to define multiplicity of edge as shown on figure 3.3.

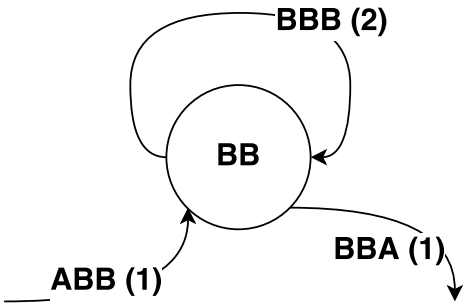


Figure 3.3: Weighted graph example.

Repeating sequences

Repeating sequences creates very similar edge types as in previous problem. But in this case the cycles are created between different nodes. The solution is similar to previous – duplicated edges are replaced by one edge with corresponding weight as shown on figure 3.4.

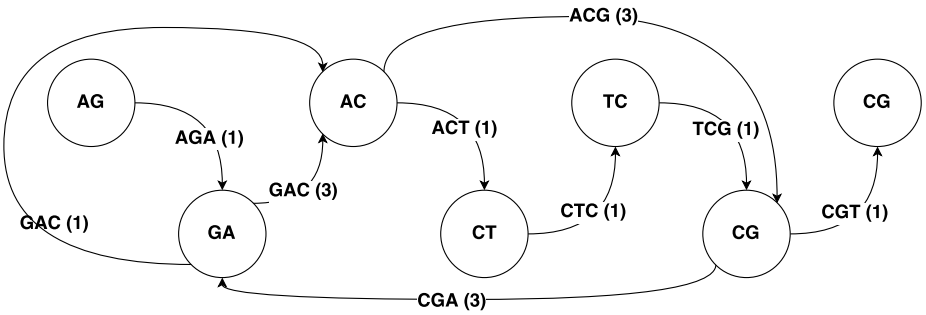


Figure 3.4: Weighed de Bruijn graph with repeating sequences. Input sequence with repetition $S = AGACTC\mathbf{ACGACGACG}G$.

Space in genome coverage

Space in genome coverage or errors of more consecutive nucleotides may produce a gap in sequence. Graph constructed will not be continuous, in other words more graphs are created in this case. This situation does not have to be problematic for purposes of thesis. In this case spanning trees are found and separate graphs are created [17].

3.2 Graph transformations - errors and their correction

Consecutive graph segments simplification

The easiest simplification that could be performed is to merge consecutive nodes into one node. While consecutive path is merged we have to keep all information that will be destroyed by merging other nodes. K-1-mer sequence of first node must be concatenated with last characters of other nodes and multiplicity of node must be summed and saved into one merged node.

- Input: de Bruijn Graph

- Output: de Bruijn Graph without consecutive segments
- Method:

```
find nodes with 2 or more inputs and 1 output
put found nodes into list of nodes
for each node in nodes:
  nodes_to_merge.append(node)
  while True:
    next_nodes := get_next_node(node)
    if count(next_nodes) > 1:
      break
    else
      nodes_to_merge.append(next_nodes)

  if nodes_merge.length() > 1:
    merge_nodes(nodes_to_merge)
```

Procedure `merge_nodes(nodes_to_merge)` preserves first node, merged information about sequence and coverage to the first node and deletes the rest of nodes.

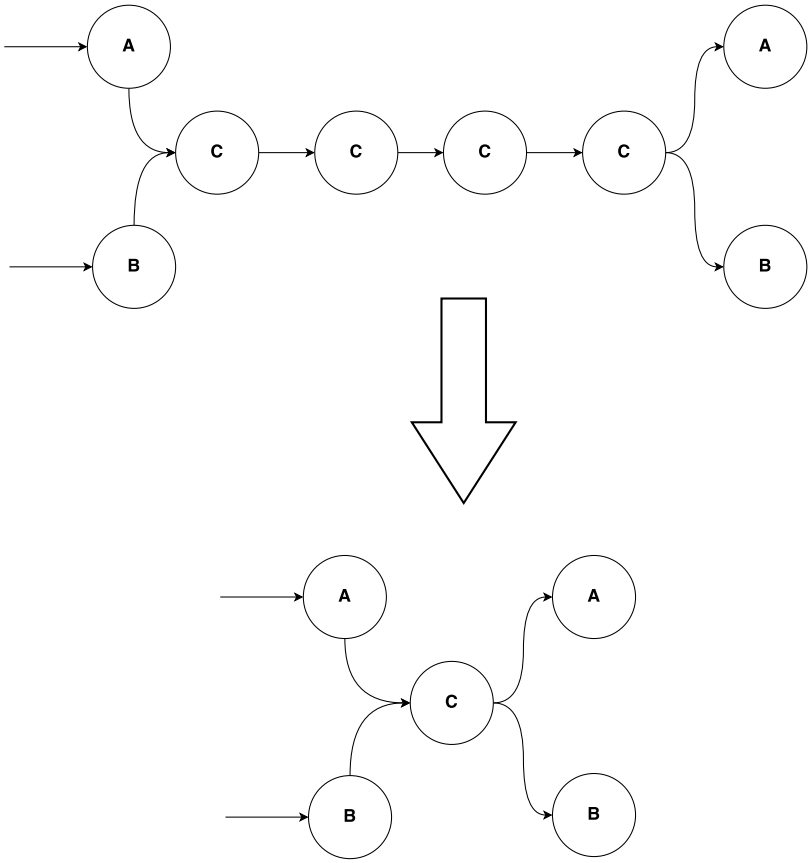


Figure 3.5: Example of simplification process. Nodes with label C are collapsed into one node.

Time complexity

Time complexity can be computed from following parts of algorithm :

- Finding nodes with special property defined in algorithm must traverse all nodes in graph – complexity is $\mathcal{O}(n)$ where n is number of nodes in the graph.
- In the next part of algorithm, there are two loops. First loop iterates over nodes with special property, second loop iterates over whole consecutive segment. So the complexity is $\mathcal{O}(m * p)$ where m is number of nodes with special property and p is number of node in consecutive segment – m is expected to be very small in comparison with p.

Space complexity

Algorithm requires to store list of nodes with special property defined in algorithm and list of nodes to be merged. In the worst case, there are all nodes of de Bruijn graph in list of nodes with special property for graph with one consecutive segment of all nodes. In the next case many nodes with special property are found with less nodes in the list of the merged nodes. Thus, algorithm have linear space complexity – $\mathcal{O}(n)$.

Input data in form of genome reads usually contain sequencing errors or SNPs (Single nucleotide polymorphism). There are several types of errors according to which place of reads is affected.

Tips removal

Tips are created by error that occurs at the end of the read and are represented as a path in graph with one disconnected end. These paths are simply deleted – tip is removed, if it is shorter than $2k$. The number was chosen because it is higher than length of k-mer in short reads. Longer sequence may be considered either as a genuine sequence or as a sequence with accumulated errors. Distinguish these two situations is difficult [25]. Example of tips removal is displayed on figure 3.6.

- Input: de Bruijn Graph
- Output: de Bruijn Graph without tips
- Method:

```
find nodes with only one input or output
put found nodes into list of nodes
for each node in nodes:
  tip.append(node)
  while True:
    next_nodes := get_next_node(node)
    if count(next_nodes) > 1:
      break
    else
      tip.append(next_nodes)

  if tip.length() < 2*k
    delete tip
```

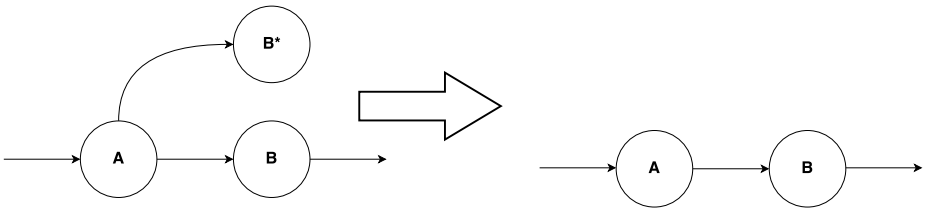


Figure 3.6: Tips removal.

Time complexity

Algorithm time complexity can be computed from following parts and is very similar to the complexity of consecutive segments merge algorithm:

- As in algorithm for consecutive segments merge, nodes with special property defined in algorithm must be found, that leads to $\mathcal{O}(n)$ where n is number of nodes in the graph.
- Iteration over nodes with special property, with inner loop that iterates over whole tip. So the time complexity is $\mathcal{O}(m * p)$ where m is number of nodes with special property and p is number of nodes in particular tip – m is expected to be very small in comparison with p.

Space complexity

Algorithm requires to store list of nodes with special property defined in algorithm and list of nodes in the tip. Count of tips is equal to the count of the nodes with special property. Thus, algorithm have linear space complexity – $\mathcal{O}(n)$.

Bubbles

If an error is present in the middle of reads typical bubble is created. In a simple way we can remove nodes and edges with low coverage that create bubbles. In fact, the process of bubbles removal is very complex and in order not to lose any information it is required non-trivial approach. Implementation are described in next chapter 5 [17].

Bubbles detection algorithm

Bubbles can be detected by using Tour bus algorithm. This algorithm is denoted as Dijkstra-like Breath first search. When a bubbled is detected, we must consider which path will be deleted and which one will be left. The whole process is described below [25].

- Input: de Bruijn Graph
- Output: de Bruijn Graph without bubbles
- Method:

```
pick an arbitrary node
until all nodes are not visited:
  #start BFS algorithm
  traverse graph with BFS
  node.visited := True for every visited node
  during BFS check:
  if node.visited == True:
    paths := paths_to_common_ancestor(node)
    merge_paths(paths)
```

There are two significant procedures in bubbles removal algorithm: `paths_to_common_ancestor(node)` and `merge_paths(paths)`. Procedures perform non-trivial traverse through graph and more detailed explanation follows.

- `paths_to_common_ancestor(node)` - If this procedure is called, it means that bubble was found. Procedure starts at given node and backtracks to the nearest common ancestor. It tries to find all possible paths that lead to the ancestor. If the nearest commons ancestor of all paths is reached, save paths.
- `merge_paths(paths)` We must consider which paths will be deleted – we must compute which path is more reliable and has higher coverage. This procedure decides which path is the most reliable and deletes the others. The metric R is used to determine the best path: distance between two vertices is divided by the sum of coverage (number of k-mers) of all consecutive forward edges between A and B. The metric is given by following equation:

$$R(A, B) = length(A, B)/multiplicity(A, B)$$

where we find the most reliable path with higher coverage from vertex A to vertex B [25].

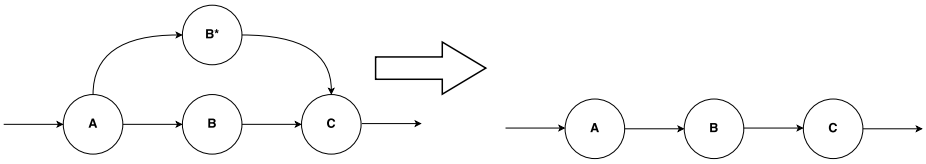


Figure 3.7: Bubbles removal.

Time complexity

Determine algorithm time complexity can be more difficult in this case.

- BFS part – time complexity is $\mathcal{O}(|V| + |E|)$ where $|V|$ is number of vertices and $|E|$ is number of edges [16].
- Paths to commons ancestor finding – complexity depends on number of paths p and their length l : $\mathcal{O}(p * l)$. After consecutive segments merge l is expected to be very low.
- Merge paths – metric for each path is computed in linear time, then only the most reliable path is kept and others are deleted. Time complexity is $\mathcal{O}(p * l + n)$ where $p * l$ s sum of nodes of all paths and n is number of count to be deleted.

Space complexity

Space required for BFS algorithm is $\mathcal{O}(|V|)$ where |V| is number of vertices [16]. Then we require to store all nodes n of currently computed paths – $\mathcal{O}(n)$.

Chimeric edges - chimeric features

A low coverage edge that connects two continuous graphs with high coverage is most probably chimeric (fake) edge. Removing that edges with corresponding node solves the problem (figure 3.8) [17]. This type of connection does not form any recognizable loop or structure and cannot be easily find as tips or bubbles. A simple coverage cutoff is used in this case. Nevertheless, coverage cutoff value is very sensitive parameter, if set too high, global cutoff process may destroy nodes with valuable information. Also global cutoff must be used after bubbles removal. During bubbles removal process we want to keep unique regions with low coverage [25].

- Input: de Bruijn Graph, cutoff parameter
- Output: de Bruijn Graph without chimeric edges
- Method:

```
for node in graph.nodes
  if node.forward_edge_coverage <= cutoff:
    delete node
```

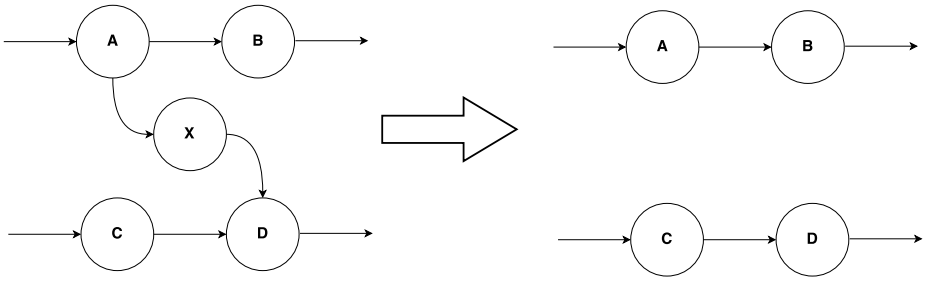


Figure 3.8: Chimeric connection removal

Time complexity

Time complexity of the algorithm is linear – $\mathcal{O}(n)$ where n is number of nodes in the graph because it traverses all nodes once.

Space complexity

Algorithm does not require any additional space. It uses space of already constructed de Bruijn graph.

Design and implementation of graph transformation algorithms is comprehensively described in the next chapter 4 and 5.

Chapter 4

Program design and architecture

This chapter comprehensively describes design of program components one by one. Each component's inputs, outputs and functionality are proposed. Program is divided into 5 components, each of them has specific functionality. At the end of the chapter there is diagram of complete functionality which is implemented in this thesis.

4.1 Architecture overview and components design

The program design and architecture leads to the goal of thesis – it is able process input data (sequence, reads) from which de Bruijn graph is created. Then the graph is transformed by several simplifying and error removal algorithms. Final graph is visualized and repeats are reconstructed. Logically the program is divided into 5 following components:

- **Input reader** – reads input data from file.
- **De Bruijn graph creator** – constructs de Bruijn graph from given reads.
- **Transformation component** – simplifies and transforms the graph by several methods.
- **Graph renderer** - visualizes the graph.
- **Repeats reconstruction component** - investigates graph structure and reconstruct the repeated sequences.

Input reader

This component is responsible for reading input data and passing data to the De Bruijn graph creator component. Only several options of input format are supported. Data are read from file or standard input. Output of the reader are reads or sequence that are used during next phase – de Bruijn graph creation. There are several input methods required, if not specified otherwise, one strand of DNA is expected on input:

- **Sequence of genome** – In this case, sequence is chopped into reads by defined parameters such as length of reads and count of reads -– this also specifies how much reads cover the sequence.
- **Reads of DNA from sequencing machine** – If reads are passed to input, it is not required to change them at all. They can be right away passed to de Bruijn graph module. FASTQ format is supported [5].

- **File with real repeats** – RepBase format [13] is expected and this special case is used for creation testing sequence and reads. Reader is able to read the sequence of repeat, its name and identification.

De Bruijn graph creator

Input of this component are reads (list of reads), which will be chopped into left and right k-1-mers. Output is supposed to be one or more (list) de Bruijn graphs that will represent input reads (sequence).
Designed functionality:

- **Chop read into k-1-mers** - Each read is processed and is chopped into left and right k-1-mers from which graph will be created in next step.
- **Build de Bruijn graph** - Graph is represented by hash table (associative array) where key is hashed left k-1 mer. Each key is mapped to lists of adjacent right k-1-mers and left k-1-mers. By this way the graph can simply be represented and created. To be more specific vertices of graph are the keys and edge is represented by each couple key-value (left k-1-mer, right k-1-mer). Each vertex has to store several necessary information – at least coverage and sequence.
- **Split whole graph into sub graphs** - In case we have very poor coverage of original sequence, as a result disconnected graph is created. Thus, it is required to split disconnected graph into connected sub graphs. Each of sub-graph is handled in the next phase of process separately.

Graph transformation module

The main goal of this module is to transform input into graph having much less vertices and edges. The graph is processed by several transformation algorithms that simplifies graph and remove possible errors. Them the output graph is easily and quickly rendered and carries as much information as it is possible. There is a risk that during transformation algorithms some amount of information is lost and therefore it is not possible to recover them. Information lost is excepted when processing complex graphs. Transformation methods are in chain that can be iteratively repeated. Every transformation method implements traversal through graph with specific actions.

Design of graph transformation methods:

- **Simplify consecutive segments of graph** – This method finds consecutive sequence of vertices in graph and merges those vertices into one. Information about sequence and coverage from merged vertices is preserved. Also number of merged nodes is kept – that information also with sequence helps us to check original sequence with the found one.
- **Remove tips** – As mentioned in chapter 3, several errors can be done during sequencing. If an error is placed at the start or end (+/- k), typical structures — tips are created in de Bruijn graph. These tips can be easily removed by algorithm mentioned in chapter 3.

- **Remove chimeric connections** – Chimeric edge between two vertices has typically very poor coverage. That kind of error is removed by global cutoff – every edge is removed, if its coverage is below defined level. Global cutoff is very sensitive operation because we can possibly lose important information but it is necessary to transform graph into simpler representation for render and next transformations.
- **Remove bubbles** – Creation of bubblse (bulges) is next typical problem discussed in chapter 3. Removal is handled by Tour-bus algorithm that is Dijkstra-like Breadth First Search. Algorithm finds bubbles in the graph and preserve the best path found, other paths are deleted.

Visualization module

For correctness confirmation and easier view of created graph, rendering module is required. Input is one de Bruijn graph, output is directed (or undirected) graph rendered to typical format (.png, .pdf . . .). During rendering process, we have to keep in mind that it is necessary to distinguish between edges with low and high coverage -– this is solved by changing thickness of the edges. Also we care about how many vertices are merged into one vertex during transformation process – it is possible to change size of each vertex according to amount of merged nodes. Amongst the graphical information, it is required to show text information about coverage, merged nodes also about type of sequence that is covered in vertices – as well as different assigning different colours to each sequence type - we much require such information during testing and evaluation process.

Repeats reconstruction module

The main responsibility of this module is to find vertices in graph that are suspected to be repeats. Input for this module is transformed graph or list of graphs, and output is list of suspicions nodes and theirs sequences.

Process is handled by following method. First, we require information about average coverage of the graph. Then we traverse through graph and if coverage of current edge is higher than average coverage, then correct vertex will be stored. Until coverage does not fall below the average, store successive vertices. After this process is finished, it is required to extract their sequence. During testing and evaluation, we need also to extract type of sequence that will be consequently compared and its sequence aligned with original sequence.

4.2 Scheme of program architecture

On figure 4.1 you can see complete overview of communication between each module with inputs and outputs. We can see that Graph visualization component is independent, as input it could take de Bruijn graph in any form - non-transformed and transformed.

Notes for testing design

The input reader component handles also reading the **testing sequence with flags** – For testing purposes reader is able to read sequence where each base has special flag. By this flag, program easily identify where is the base from or what type is it. That is either repeat (each marked with unique identifier) or unique (random) sequence. More about

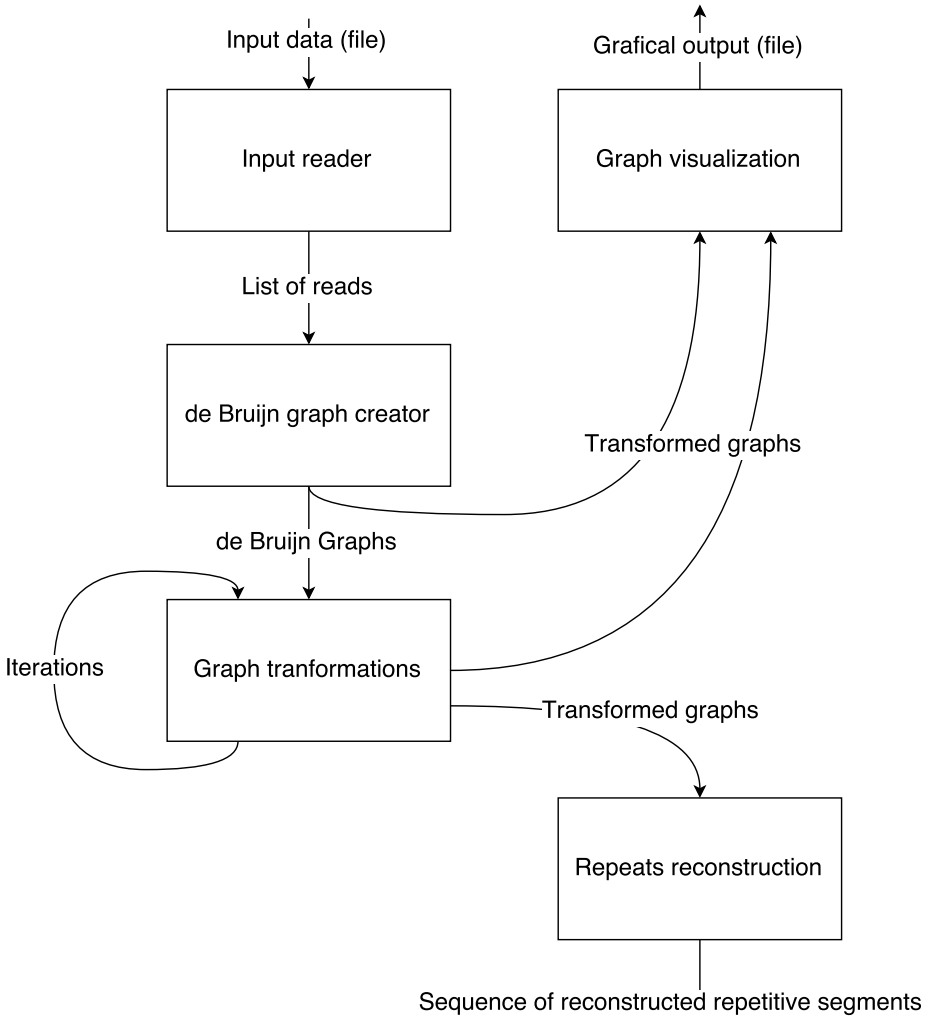


Figure 4.1: Diagram of program architecture.

creating of testing sequence and testing itself along with design of sequence generator is specified in chapter 6.

Chapter 5

Implementation

Chapter provides description of implementation of designed program. Implementation of each component proposed in chapter 4 is provided with necessary details and references to the source code – names of functions, methods and classes. Input reader module implements necessary functions for input processing. De Bruijn graph and graph transformation components are consolidated to the one class/module because of their logical relationship. Description of the rest of components for graph visualization and repetitive segments reconstruction is at the end of chapter.

5.1 Input reader

Component for input reading is implemented in `InputReader.py - class InputReader`. There are two methods which perform simply read from file:

- `read_sequence()` - Read the whole sequence from give file. The sequence is subsequently chopped into reads.
- `read_reads()` – Method expects one read on each line of file or FASTQ [5] file format can be used as input.

Input for testing and evaluation:

- `read_testing_data()` - Method reads data for testing purposes, data must exactly formatted and included sequence itself and also flags of each base that determine type of sequence.
- `read_repeats()` - Read data from RepBase [13] file. Again this method is used for testing sequence construction.

Detailed explanation testing data creation and storage is provided in chapter 6.

5.2 De Bruijn graph and transformation components

De Bruijn graph creator and graph transformation components are implemented in `DeBruijnGraph.py`. They are implemented one module because of logical relationship. Module includes `class DeBruijnGraph` with inner `class Node`. The purpose of this module is to construct de Bruijn graph as well as conduct transformation methods.

de Bruijn Graph

Graph itself is stored in field `G` – it is a dictionary that contains pairs key-value (*dict* Python built-in type). Keys of dictionary are left k-1-mers (hashed k-1-mer string) and value consist of two lists - list of successors - right k-1-mers and a list of predecessors - left k-1-mers. Despite that bidirectional implementation of de Bruijn graph storage have bigger memory consumption, it is very useful for simplifying and traversal algorithms (forward and backward movement) used in program. There is another dictionary in the class - dictionary of nodes that maps k-1-mers to actual Node objects.

Static generator method `chop()` yields quintuple that consists of strings k-mer, left k-1-mer and right k-1-mer and for testing purposes there are yielded two lists of sequence type[1] of left and right k-1-mer. `class Node` represents k-1-mer itself, most significant fields are sequence – bases of node (string), number of inputs and outputs to unique nodes and coverage of k-1-mer.

Construction of graph is implemented in constructor of `class DeBruijnGraph` which takes iterator object of string (represents list of reads) and `k` integer as argument for k-mer length. Constructor iterates string iterator in for loop and each string (read) is processed separately. Then every left and right k-1-mer are appended to the nodes list, if they were not present there previously. Left (right) k-1-mers are added as keys to the dictionary and right (left) k-1-mers are appended to the list of successors (predecessors) list. Those lists contain only unique k-1-mers and their multiplicity is stored as coverage field of each node. Example dictionary of de Bruijn graph is displayed on figure 5.1.

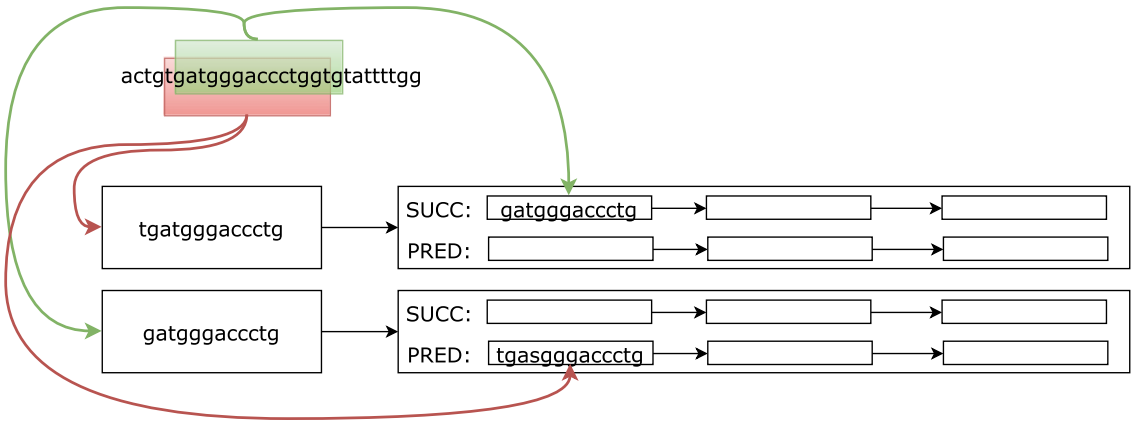


Figure 5.1: Example of de Bruijn graph data structure - left (red) and right (green) k-1-mers.

Sub graphs

Because many disconnected graphs can be created during de Bruijn graph construction, that single disconnected graph is divided into sub graphs – each of them is connected. Algorithm uses *Breadth First Search* method to traverse whole graph and find spanning trees of sub graphs. It is implemented in method `find_subgraphs()` in `class DeBruijnGraph`. Method iterate over all nodes in graph and checks whether node is in spanning tree of any sub graph (boolean flag `inSpanningTree`). If not, graph is traversed by BFS and method finds another spanning tree – sub graph.

[1]Sequence type means what type the current bases of k-1-mer are

Graph transformations

Algorithms for graph transformation are also implemented in `DeBruinGraph.py` module. They are logically connected to de Bruijn graph, thus they transform given graph in place. There are 4 methods that handle transformations: `simplify()`, `removeBubbles()`, `remove_tips()`, `remove_chimeric_connections()`. Algorithms for simplification and errors correction are implemented according to its theoretical specification in chapter 3.

- **Merge consecutive segments of graph:** method `simplify()`
  Firstly, nodes with special property are found. That is a node with more than 1 output node and with number of input nodes different from one. Now algorithm iterates over all found special nodes one by one and it traverses adjacent nodes, saves them to list of node which will be merged until adjacent node with more than one input or more than 0 outputs is found. To keep graph consistency, then we must delete edge from preceding node to the first node which will be merged and add new edge from the preceding node to the last one to jump all merged nodes. It must be done also from opposite direction because of bidirectional implementation of graph dictionary. During merging of nodes we have keep information from all nodes – sequence, coverage and also number of merged nodes. Sequence is simply concatenated to the preceding node, coverage is adjusted to the average of merged nodes and preceding one and number of merged nodes is saved to preceding one. For testing purpose preceding node will also store information about sequence type of merged nodes.

- **Tips removal:** method `remove_tips()`
  In this starting nodes (ending nodes respectively) must be found. These nodes have 0 inputs (1 input) and 1 output (0 outputs). Both starting and ending nodes must be handled separately – traversal benefits from bidirectional implementation of de Bruijn graph. Adjacent nodes are traversed until nodes with two or more adjacent nodes are encountered. If this path contains less than `(tip_lenght * k)` bases, we can assume we found a tip and delete its nodes. Tip is usually set to value 2, for explanation see chapter 3.

- **Chimeric connections removal** – method `remove_chimeric_connectionns()`
  Implementation of algorithm for chimeric connections removal is quiet simple but we must carefully delete edges from preceding and succeeding nodes of deleted one. Finding nodes with less coverage than cutoff parameter is the first part of an algorithm. Found nodes are deleted and edges from preceding and succeeding nodes are also deleted in order to preserve consistency of graph.

- **Bubbles removal – Tour bus algorithm :** method `remove_bubbles()`
  Algorithm is based on *Breadth First Search* – every time we step on any node we set its flag `visited = True`. Special situation occurs when we step on node which previously visited. The we start backtracking to the nearest commons ancestor. The method which implements backtracking is called `backtrack(node)`. Backtrack tries to find all paths to the common ancestor. First, we determine number of possible paths by number of previous of the backtracked one. Then we traverse previous nodes on each path but we must alternate between paths on every step. Nodes are flagged as `backtracked = True` and if we step again of flagged node, then common ancestor of paths is found. Search for common ancestor is finished when all paths get

their common ancestor. Method `backtrack(node)` returns dictionary containing all possible paths from backtracked node to the nearest common ancestor.

All found path must be merged into one by method `mergePaths()`. We must decide which one has the highest coverage – the metric that decides is mentioned in chapter 3. The path with the best properties will be kept, other ones are deleted. Graph must stay consistent after paths removal so references to successor of common predecessor and references to predecessor of common ancestor are also deleted.

5.3 Visualization module

Module is implemented in `plot.py` and uses *igraph* [2] library for graph rendering in order to allow visual analysis.

First of all, dictionary graph representation must be converted to igraph representation. Implementation is in `class Plot` and method `createIgraph()`. There are also two variants of graphs, we can create non-oriented version but also oriented one which gives us better understanding of what we can see on graph plot. For creating of igraph, firstly we create vertices by iterating over keys in de Bruijn graph dictionary, secondly we create edges by iterating over key-values in dictionary where key is source node and values consist of destinations nodes.

Size of nodes is adjusted according to the number of merged nodes during transformation process – size is normalized to maximal and minimal count of merged nodes in graph. Each node has also label consisting of sequence type and number of merged nodes.

Width of edges is set according to coverage and is normalized to maximal and minimal coverage in given graph. Graph layout is derived from count of nodes in graph by `igraph method layout_auto()`. If the graph is connected and has less than 100 nodes, then the *Kamada-Kawai* layout is used. If the graph has less than 1000 vertices, then the *Fruchterman-Reingold* layout used. Otherwise the *DrL* layout is used [2]. Because it takes a very long time to plot the complex graphs with many vertices (10000 or more), graph is plotted, if count of vertices is lower than 10000. Even if we can wait to render such complex graph, result of visualization is not reasonable and it is very difficult to analyze the structure.

5.4 Reconstruction of repeats

Module from repetitive segment reconstruction is implemented in `Reconstruction.py`. Reconstruction is based on coverage of nodes. As a results of reconstruction we get list nodes which most probably consist of a repeat, if there are more repeats in a graph, we get more lists of nodes – each list will hold a repeat.

First, we must find starting nodes (for definition see section 5.2). Algorithm iterates over those nodes and for each of them *Depth First Search* algorithm is executed. During DFS we are looking for nodes with coverage higher than average. Once we step on such node, we start appending nodes to the list until we step on node with lower coverage than average.

Alternate paths

It possible that there are more alternate path in graph that form one repeat – it happens because of several reasons – but mostly it occurs when complex structure of graph is present that cannot be eliminated by graph transformations. In this case we must consider that all paths can comprise repetitive segment. During algorithm when the beginning of an alternative path is found, all previously paths found so far are duplicated[2] and now all paths are filled with next nodes until algorithm ends. Output of this final step is sequential number and sequence of single repeat. Pseudo-code of repeats reconstruction follows.

- Input: transformed de Bruijn graph
- Output: sequences of reconstructed repetitive segments
- Method:

```
for each starting node in the graph:
  start DFS algorithm
  path = []
  if node.coverage > avg_coverage:
    in_repeat = True
  else:
    in_repeat = False
    output << get_sequence(path)

  if in_repeat:
        path.append[node]

  if get_next_nodes(node) > 1:
        duplicate_paths(paths)
```

5.5 Used technologies

Program is implemented purely in Python programming language (2.7.11, 64bit) [9]. Python is interpreted and object-oriented language and is widely used in research and development in bioinformatics. Program written in Python are portable to many platforms. For graph plotting igraph library for Python [2]. This library is portable and is implemented in R, Python and C programming languages. By means of the library – graphs can be created and manipulated and also analyzed – igraph library implements tools for graph and network analysis, creation, manipulation and plotting. In this thesis, igraph is used mostly for plotting the de Bruijn graph. Igraph library uses *Cairo* library to render graphs [1]. BioPython package is used for parsing FASTQ file format [3]. That package provides many useful functions for reading and writing data in standard biological format and can be used for sequence manipulation as well. Program was developed using PyCharm IDE [8] and source code version system GIT [6] was used with private repository on BitBucket [4].

[2] Copies count of current paths depend on number of alternate paths

Chapter 6

Testing and evaluation

This chapter provides complete explanation of testing and evaluation of created program. In the beginning of chapter, implementation of sequence generator for testing and its evaluation is explained. It is followed by tests of graph transformations. Test scenarios were mainly focused to determine the scale of graph transformation. We measured count of edges and nodes before and after transformation. Also errors correction was tested, especially bubbles detection and elimination as this problem is very complex. We showed that all transformations lead to simpler graph without any information loss and that the final graph is easily visualized. Next part consists of repeats visualization and evaluation of reconstruction on given sequence.

6.1 Genome generator

For testing purposes, genome generator was developed. The goal is to create sequence with reference of every base in that sequence. Therefore each base has own reference that describes its type. References are essential for further analysis and testing of graph transformations as we require knowledge of what type of bases are hidden in every node in the graph otherwise we would not be able visually determine correctness of transformations. Distinguishing between sequence types is used especially in visualization when different colors can be assigned to each node accordingly to the sequence type.

Types of base of sequence are divided into two groups:

1. Inter genomic sequence that is represented by randomly generated sequence. In the text below this type of bases are referenced as *random bases* or *random sequence*.

2. Particular repeat with sequence and type from actual genome of organism.

All methods for sequence generator are implemented in `SequenceGenerator.py`. An approach that creates a **list of positions** of each random or repeated sequence in the final genome was implemented. Each item of that list contains starting and ending position and sequence data – sequence and type of sequence (random sequence or particular repeat).

Basic approach is to create random sequence of length that is derived from final genome length and repeats filling scale (explanation follows below). Then insertion of sequences with repeats is performed until final length is reached. The chain of whole genome generator is displayed on figure 6.1.

Genome generator require several initial parameters and input data - actual repeats. Those repeats are iteratively inserted into genome. Insertion could be executed in several modes.

Initial parameters and data for sequence generator:

- **Final genome length** — represents the final length of genome that is generated.
- **Repeats filling scale** — defines what part of final genome is represented by repeated sequences. Parameter is given in percent. E.g. if repeats filling scale is set 80%, then 80% of genome contains repeats, the rest (20%) is represented by random sequence.
- **Length of initial random sequence** — used at the beginning of the sequence creation. At the start of algorithm it is represented by single item in list of sequence positions. During sequence generation random sequence is typically split into many smaller segments. The length is derived by final genome length and repeats filling scale parameters.
- **Mutation rate** - how many bases are mutated in each inserted repeat - given in percent.
- **Input data** - sequences of repeats of any organism given in *RepBase* format.

Repeats insertion modes:

- **Insertion into repeat disallowed** - Repeat are not inserted into themselves. Therefore, in the final genome there are repeats of full length (no fragments). Two options of insertion are possible:
  - **Insertion with spaces** - Original sequence of repeat is prepended and appended by random sequence of given length. If insertion would be placed into another repeats, insertion is moved to the place where the start another random sequence. Between two repeats in the genome there is always some space - random sequence. Thus, repeat that immediately starts next to another repeat is not possible.
  - **Insertion without spaces** - Sequence of repeat is not extended by any random sequence. Insertion into another repeat is handled by the same way as in previous case. In this case repeats can immediately start next to another repeat, even the same repeat can start next to the repeat. Situations described creates new difficulties and must be handled appropriately.
- **Insertion into repeats allowed** - Sequence of repeat is not extended and insertion into another repeat is allowed. This mode creates the most complex genome and handling graph constructed from such genome is the most difficult. In this mode every repeat is fragmented likely. But the worst situation comes with the longest repeats (10 000 bases or more) - these repeats are highly fragmented and their reconstruction may be impossible.
- **Mutation allowed** - If mutation is allowed, before insertion each repeat is mutated at given mutation rate. Single nucleotide mutations are created - the count of mutation is derived by length of repeat and percent of mutation rate. Repeats with mutations could be inserted by different ways described above.

Sequence generation comprise several stages:

1. **Creation of initial item in positions list** – representing random sequence of given (initial) length.
2. **Insertion of repetitive segments until final length of genome is reached** – This stage can be executed in several modes described above – insertion with or without spaces, insertion into repeat allowed or disallowed.

   - **Spaces** - sequence of repeat is extended by random sequence.
   - **No spaces** - sequence is not altered.
   - **Mutation** - if mutation rate is higher than zero, sequence of repeat is created according to the mutation rate.

   (a) **Starting position** - new starting position of inserted repeat is randomly generated. Then behavior differs according to selected mode:
   - Insertion allowed - original starting position that was generated is used.
   - Insertion disallowed - in order to avoid insertion into another repeat - original starting position is moved to place where starts random sequence.

   (b) **Index to positions list** - Correct index to positions list is found by bisection search method - starting positions of index items are compared. New item is inserted to the found index of the position list.

   (c) **Inserted sequence split**
   - Starting position of new item equals to the starting position of existing item. Then positions are updated – explanation is below.
   - Otherwise original sequence must be split into two sequences – repeat is inserted between them.

   (d) **Update of positions** - Starting and ending positions must be updated after each insertion. Items that follow the inserted one are updated – starting and ending positions of the items are increased according to the length of inserted item.

3. **Assembly of final sequence of generated genome** – genome is assembled from items in the list of sequence positions. Method iterates over items in the list and writes sequence into string. If random sequence or space is in the item, sequence is created randomly. Along genome a string of types/flags of bases is also created for evaluation purpose. Bases type is encoded by the type, two-byte encoding is used ("rr" stands for random sequence, „ss" - spaces, „AA" - „ZZ" – repeats). Genome with its sequence and flags is saved to file for later use. For less memory consumption and reasonable performance **cStringIO** Python module is used for writing sequence and flags to string.

Reads Generator Input data for de Bruijn graph construction are prepared by Reads generator. Reads are generated randomly by following method:

- **Inputs:**
  - Sequence of generated genome.

  - Flags of the generated.
  - Reads number – derived from following parameters: genome final length, reads coverage and read length. Reads coverage means how much percent of genome is covered by reads.
  - Read length – uniform read length.

- **Output:** List of reads - each read consist of sequence and its type.
- **Method:**

  1. Generate random number – index – minimum is 0, maximum is `genome_final_length` − `read_length`.
  2. Chop read from genome at given index.
  3. Find information about sequence type in the genome flags of the chopped read.
  4. Append read with its sequence to reads list.
  5. Continue with point 1 until reads count is reached.

Determining value of k-mer parameter

In order to determine correct range of *k-mer* parameter various values of this parameter were tested. Only random sequence was used for tests. Test scenarios included tests for k in range from *1 – 10*, and tests with $k > 10$. As we can see on figures 6.2 and 6.3, value of k parameter should at least *10*. Smaller values creates graph with nodes that have a very big count of adjacent nodes because even in random sequence very short repeated sequences are present.

The bigger k is, more unique $k-mers$ are found and therefore the size of the dictionary for de Bruijn graph representation will rise. With big k parameter we also get more disconnected graphs. Maximum value of k parameter is equal to the length of read. Because for de Bruijn graph we use reads that are chopped into $k-mers$ there is no reason to use bigger k.

6.2 Graph transformation results

Consecutive segments merge

Graphs constructed from mutated reads contain unwanted structures as mentioned in chapter 3 – tips, bubbles and chimeric connections. Those erroneous structures are eliminated by running several transformations of graph (see chapter 5). Simple reduction of consecutive parts of graph is firstly applied to the constructed de Bruijn graph. Considerable reduction of node count is reached and difference in count nodes and edges before and after reduction is summarized in table 6.2 and displayed on figures 6.4 and 6.5. Both visualized graph store the same information, only several parts of the graph were simplified for easier visual analysis.

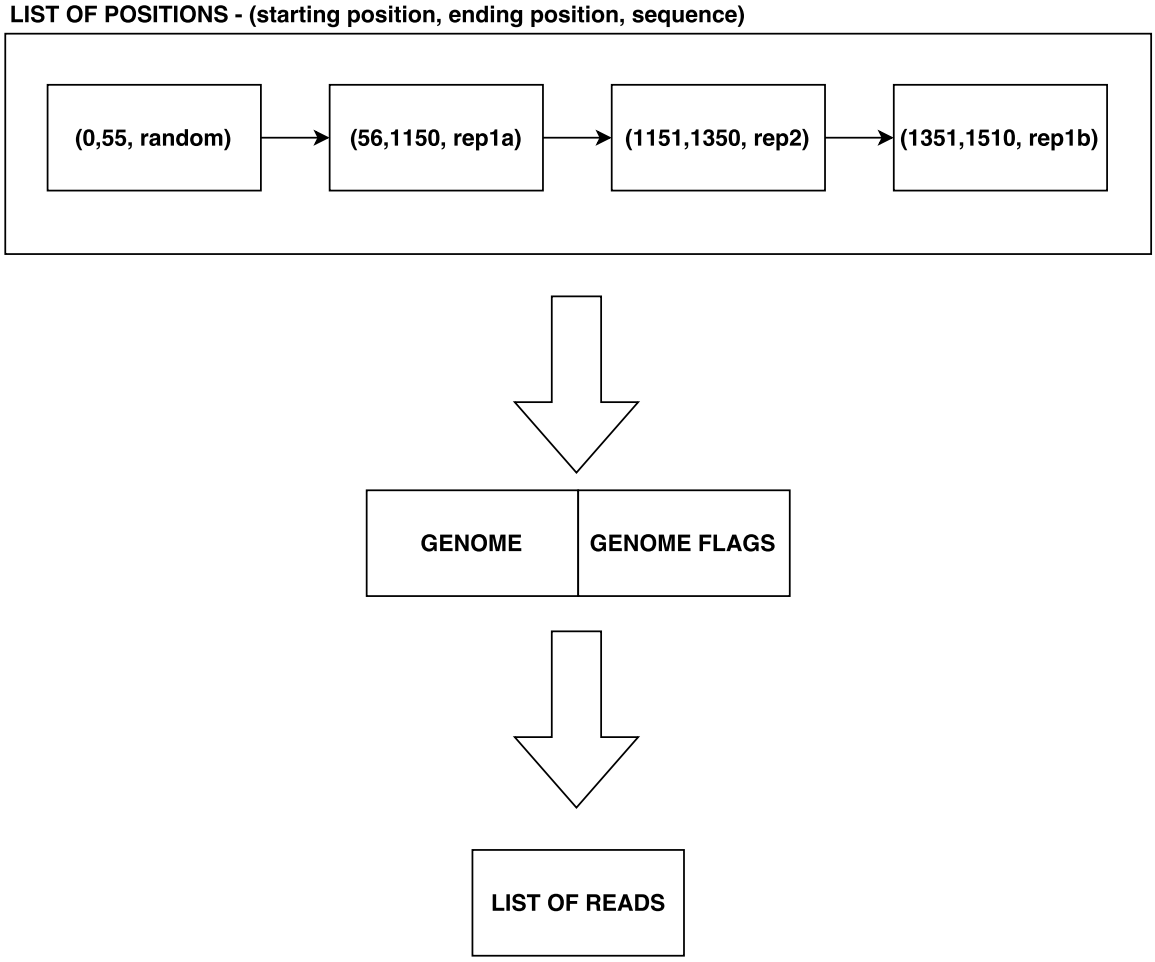


Figure 6.1: The chain of genome generator. Positions list is created at first with all required information for genome assembly - starting and ending positions of every inserted sequence and the sequence type. On figure we can find 2 repeats – `rep1` and `rep2`. Repeat `rep2` is inserted inside `rep1` which is split into two segments – `rep1a` and `rep1b`. At the start of positions list there is short random sequence. From the positions list genome is assembled with genome flags that represent type of each base. Then reads are generated and passed to de Bruijn graph creator.

	count of nodes	**count of edges**
before transformation	748	757
after transformation	74	83

Table 6.1: Comparison between count of nodes and edges on non transformed and transformed graph

Tips, bubbles and chimeric connections removal

If the graph does not contain continuous parts, we may proceed with tips removal, bubbles and chimeric connections removal. More complex mutations are now applied (each repeat is mutated with several single point mutation) and also longer sequence is generated. First tips are removed. Then by using Tour bus algorithm bubbles are eliminated. More

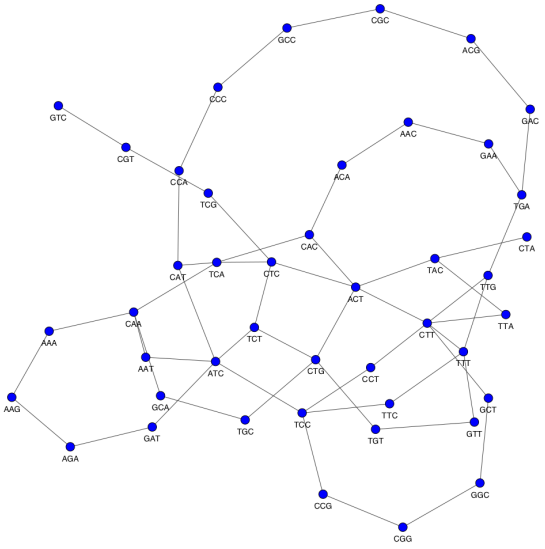


Figure 6.2: Graph with k=4 contains many cycles. Other parameters: genome length - 60, reads length - 50, 20 reads, no mutations.

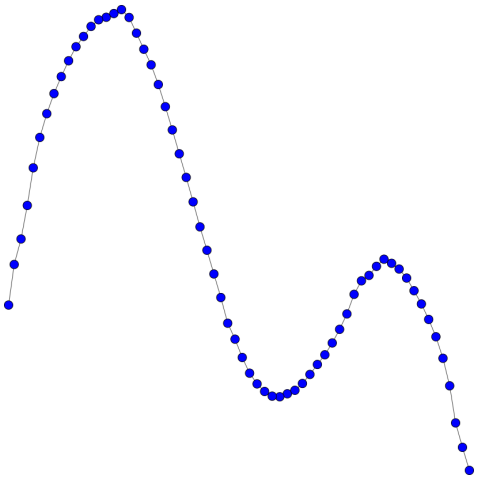


Figure 6.3: Graph with k=13 - continuous graph. Other parameters: genome length - 60, reads length - 50, 20 reads, no mutation.

complex bubble structures cannot be eliminated because there are non trivial connections between nodes which are unresolvable. But those complex connections can be considered as chimeric connection and are removed by chimeric connection removal algorithm. Tips and bubbles removal do not eliminate a lot of nodes and edges but the simplification is very useful for next iteration of transformation process. In this test I showed how successful are all transformation algorithm and final count of nodes and edges is compared with the original graph. Quality of transformation and repeats reconstructions are not analyzed - these cases are evaluated in following sections. Repeats from *Oryza Sativa* were used.

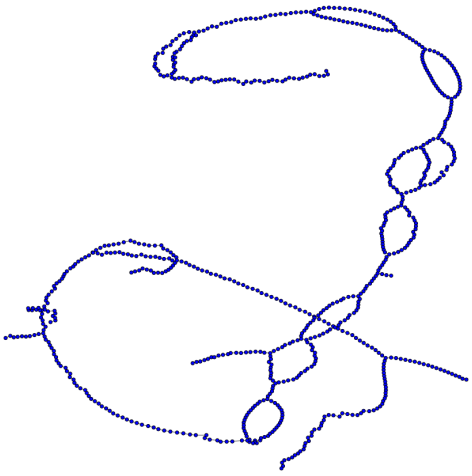


Figure 6.4: Graph with k=20 with SNPs before continuous parts reduction. Other parameters: genome length - 500, reads length - 150, 75 reads, mutation probability - 1%.

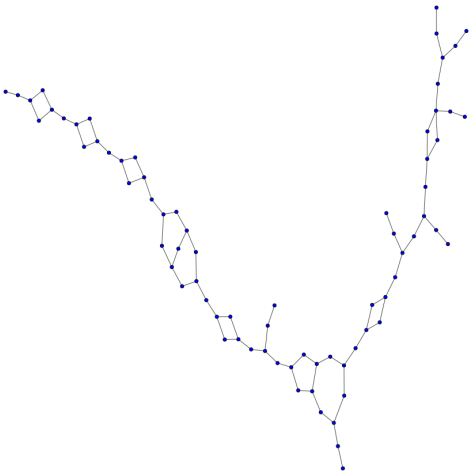


Figure 6.5: Graph with k=20 with SNPs after continuous parts reduction. Node labels are omitted. Other parameters: genome length - 500, reads length - 150, 75 reads, mutation probability - 1%.

Transformations were tested on genome with three repeats and sequence parameters are as follow:

- Repetitive segments:
  - SZ-24LTR, 428bp, LTR Retrotransposon
  - COPIA2-LTR_OS, 962bp, Copia
  - RILN8_OS, 836bp, Non-LTR Retrotransposon

- Genome length: 150000bp, k – 31
- Repeats coverage: 80%
- Reads: count – 75, length – 200bp
- Mutation rate: 0,5 %

Firstly, original de Bruijn graph was created. This graph is too complex to be visualized because it contains more than 6000 nodes. Then graph transformations are applied - consecutive segments merge - figure 6.6, tips and bubbles removal (figure 6.7). Last, the chimeric connections removal is applied with cutoff parameter 1. On figure 6.8 we can see that it is possible to transform graph again because new consecutive segments are created as result of all transformations. Thus, graph transformation can be executed iteratively. In case we create genome with lots of repeats and the final sequence is very long (50Mbp or more), chimeric features are created. In order to create graph that can be visually analyzed, we must get rid of chimeric connections and run transformation algorithm iteratively. Results of graph transformations are summarized in table 6.2.

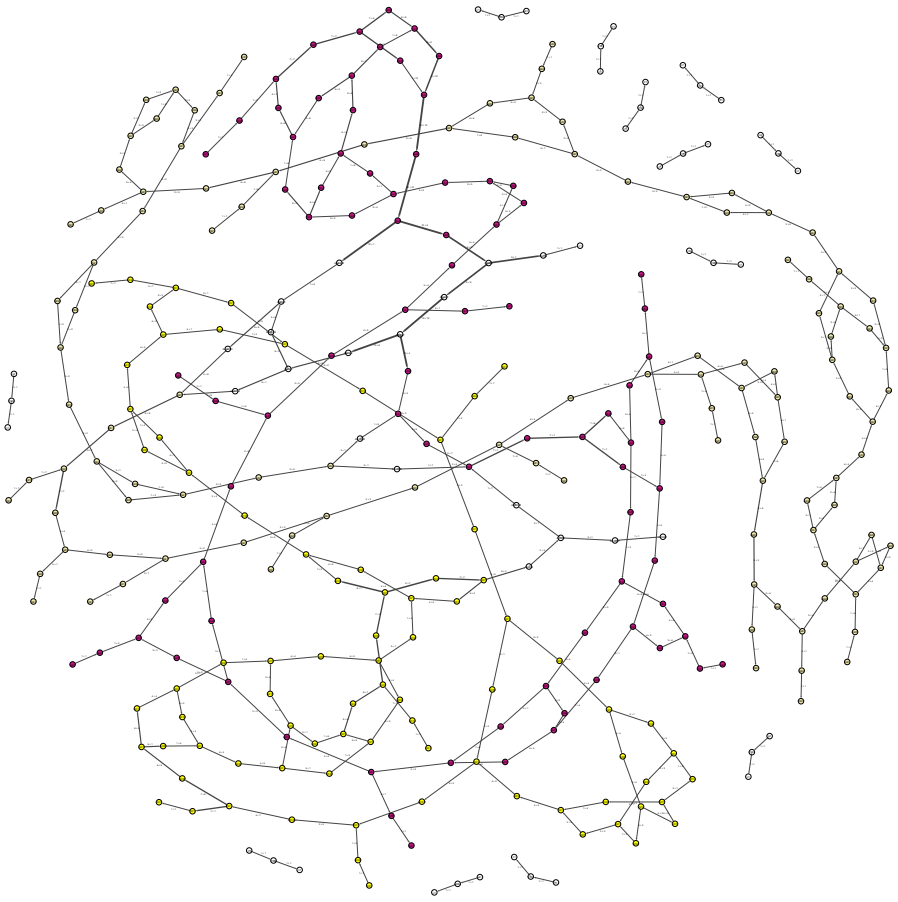


Figure 6.6: Graph visualized after consecutive segment merge. Nodes count lowered from 6027 to 331.

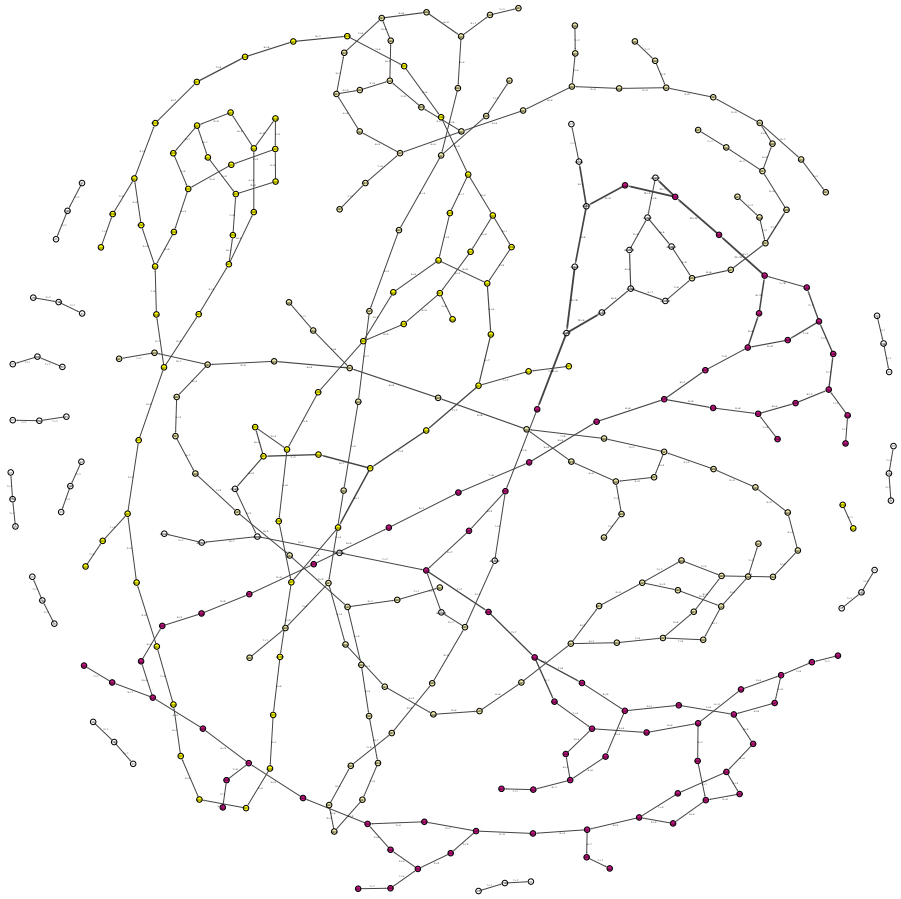


Figure 6.7: Graph visualized after tips and bubbles removal. Nodes count lowered from 331 to 312.

6.3 Repetitive segment reconstruction

In this section reconstruction of repetitive segments is evaluated. Evaluation of program is conducted from the least complex genome – small number of repeats, spaces between them, no mutation – to the complex genome – lost of repeats, insertion to repeats allowed, sequences mutated. Ability to reconstruct repetitive segment is discussed at each test case. In the first part of this section a visual recognition of particular repeats of different repeats types. Next part consists of reconstruction evaluation in several levels of genome complexity.

Testing genome Genome for testing was generated by Genome Generator described in the beginning of this chapter. Every test case has special parameters but some of genome parameters are the same for all test cases. Description of parameters follows.

Invariant genome parameters:

- Repeats coverage: 80% of genome – typical for Oryza sativa,
- Read length: 200bp.
- Reads coverage: 10% of genome.

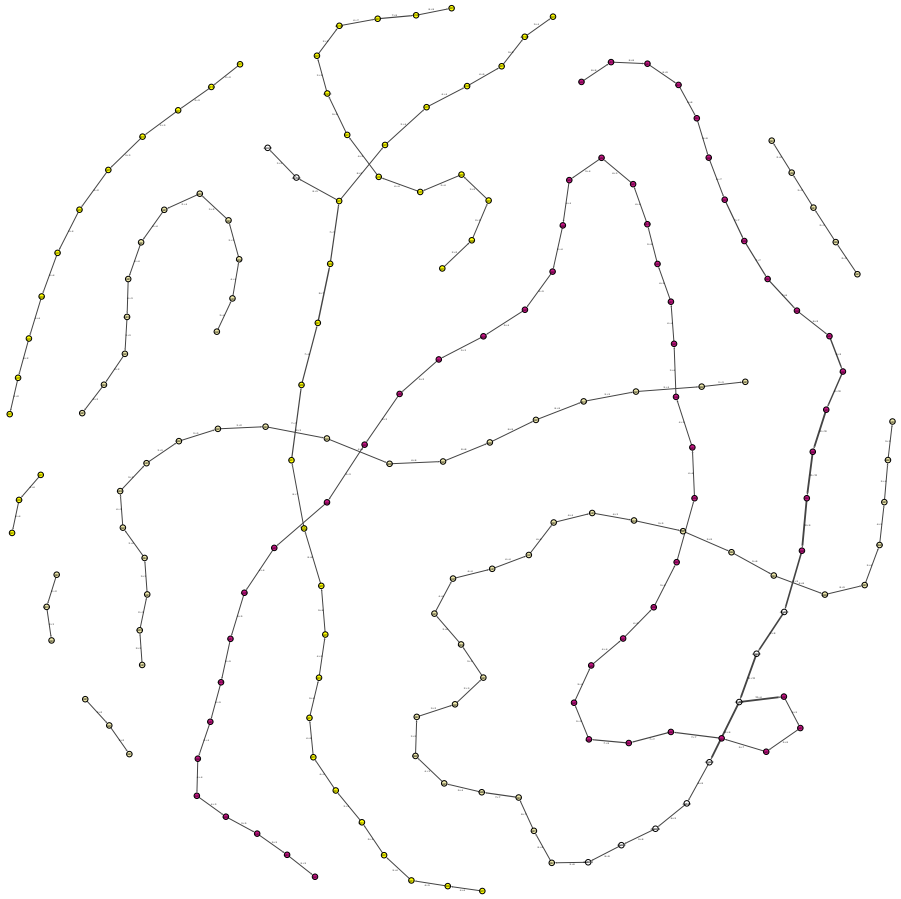


Figure 6.8: Graph visualized after chimeric features elimination with cuttof 1. Nodes count lowered from 312 to 185. We can see that next transformation are possible - new consecutive segments were created.

- Repeats data source: RepBase file of Oryza sativa repeats.
- K-mer parameter: 31.

Variant genome parameters:

- Genome length: varies from very short 0,5Mbp to 200Mbp.
- Count repeats, types of repeats: accordingly to the purpose of test cases.
- Spaces between repeats in genome: None or given length.
- Insertions: Allowed/disallowed.
- Mutation rate: None, 0,5-2%.
- Cutoff parameter for chimeric features removal.

	# nodes	# edges	percent of nodes comparison to original graph
original graph	6027	6053	100,0 %
consecutive segments merge	331	357	5,5 %
tips removal	329	355	5,4 %
bubbles removal	312	324	5,1 %
chimeric connections removal	185	175	3,0 %

Table 6.2: Summary of error removal results. Significant decrease of nodes and edges was achieved - visual analysis of constructed graph is possible and plot of graph takes less time.

Direct repeats recognition

In this test case I would like to show visualization of single repeat in graph and its reconstruction. Genome (0,5 Mbp) with spaces between repeats contains only one repeat – *COPIA2-LTR_OS* - Copia tansposon (962 bp). There is no mutation of inserted repeats. If the sequence of repeat is direct – is it expected that it does not have any satellites inside or any sequence that might result into more complex graph. Repeats with special internal structure are discussed in another section – repetitive segment with special internal structure. After several graph transformations – consecutive segments merge, tips and chimeric connections removal single repeat is visualized as three vertices in the graph. Vertex in the middle represents almost the whole repeat with high count of merged nodes. It contains inner section of repeat sequence without ends. Vertices at beginning and the end are borders of the repeat that are connected to the unique (random) region. Also the edges have high coverage which will help with repeat reconstruction. Graph visualization is displayed on figure 6.9. Repeat is distinguished by colour of node but what is more important, high number of merged nodes results into bigger nodes and higher coverage is differentiated by thicker edges. This helps repeat to be easily visually recognized. Reconstruction of such simple repeat is flawless – whole repeat is reconstructed with correct length and sequence.

Satellites

Satellites insertion Satellites are different from other repeat because their base sequence is multiply copied after itself. Insertion of such sequence is corrected. From input data we get only the base sequence – thus, before insertion that base sequence must be concatenated with itself many times.

Satellites recognition This test case demonstrates visualization of satellites and its reconstruction. Genome (10Mbp) with spaces between repeats contain satellite – *TRSA_OS* (353bp) and transposon. Base sequence of satellite of was copied 100 times and this large sequence was randomly inserted. Satellite is visualized as several nodes connected in circle with extremely high coverage. Because the base sequence is repeat many time in the whole genome. That high coverage may misrepresent results of repeats reconstruction, because the average of coverage is too high – basic repeats are omitted. If graph of satellites and other repeats are separated, the problem is solved – each sub graph have different average coverage. Every node may have also connection to the nodes of random sequence. Satellite visualization is displayed on figure 6.10. Satellite is clearly distinguished by size of nodes

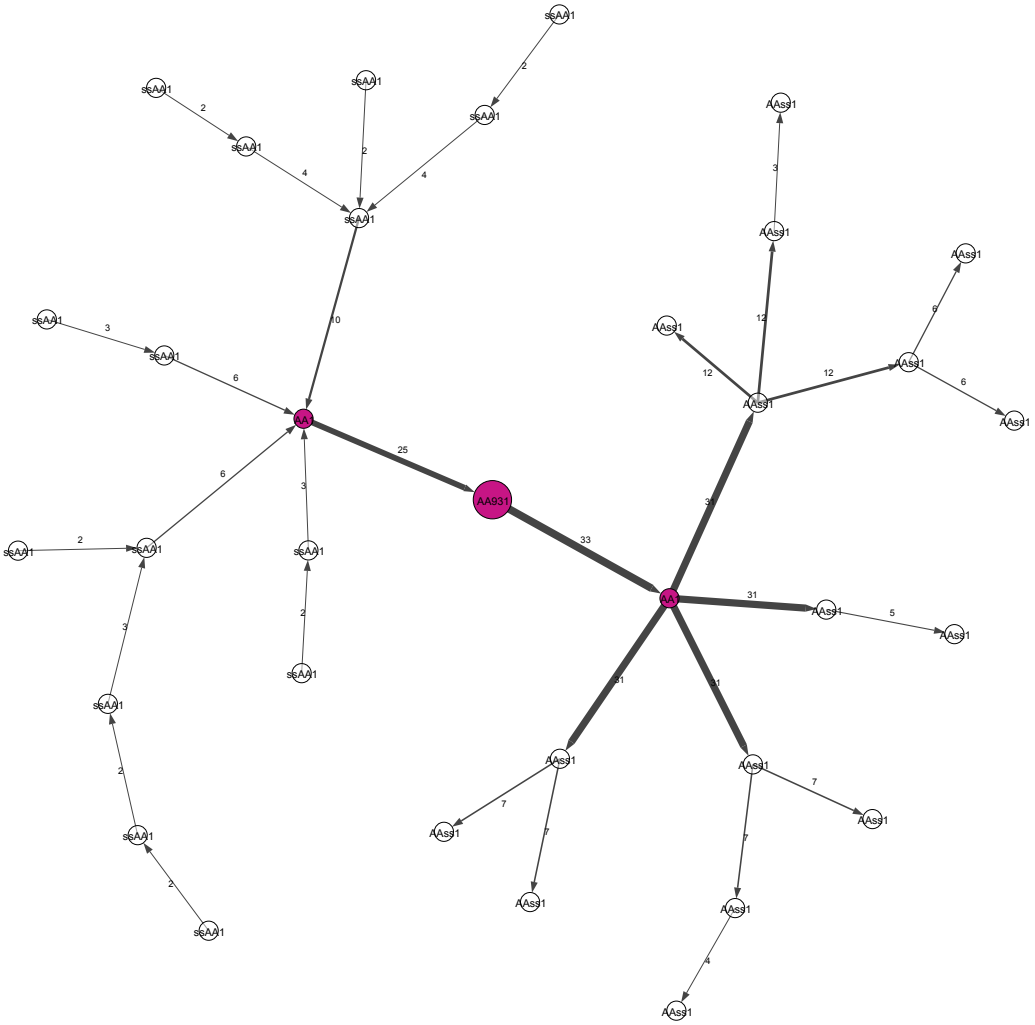


Figure 6.9: Visualisation of single repeat in genome. Pink nodes contain only sequence of repeat – middle node represents inner sequence of repeat, the other two nodes represent ends of repeat which are connected to the random sequence. White nodes stand random and repeat sequence combined. Reconstruction phase finds nodes in pink and stores their sequence.

(number of merged nodes) and especially by extremely thick edges. Algorithm is able to reconstruct the base sequence of satellite.

Repeats with non trivial sequence

During testing phase of the thesis I have encountered several repeats with non trivial (direct) sequence. Their sequence contains micro-satellites inside or some smaller segments are duplicated with mutations. Such repeats are very difficult to reconstruct because of their unusual structure. I chose several repeats with unusual structure and analyzed them.

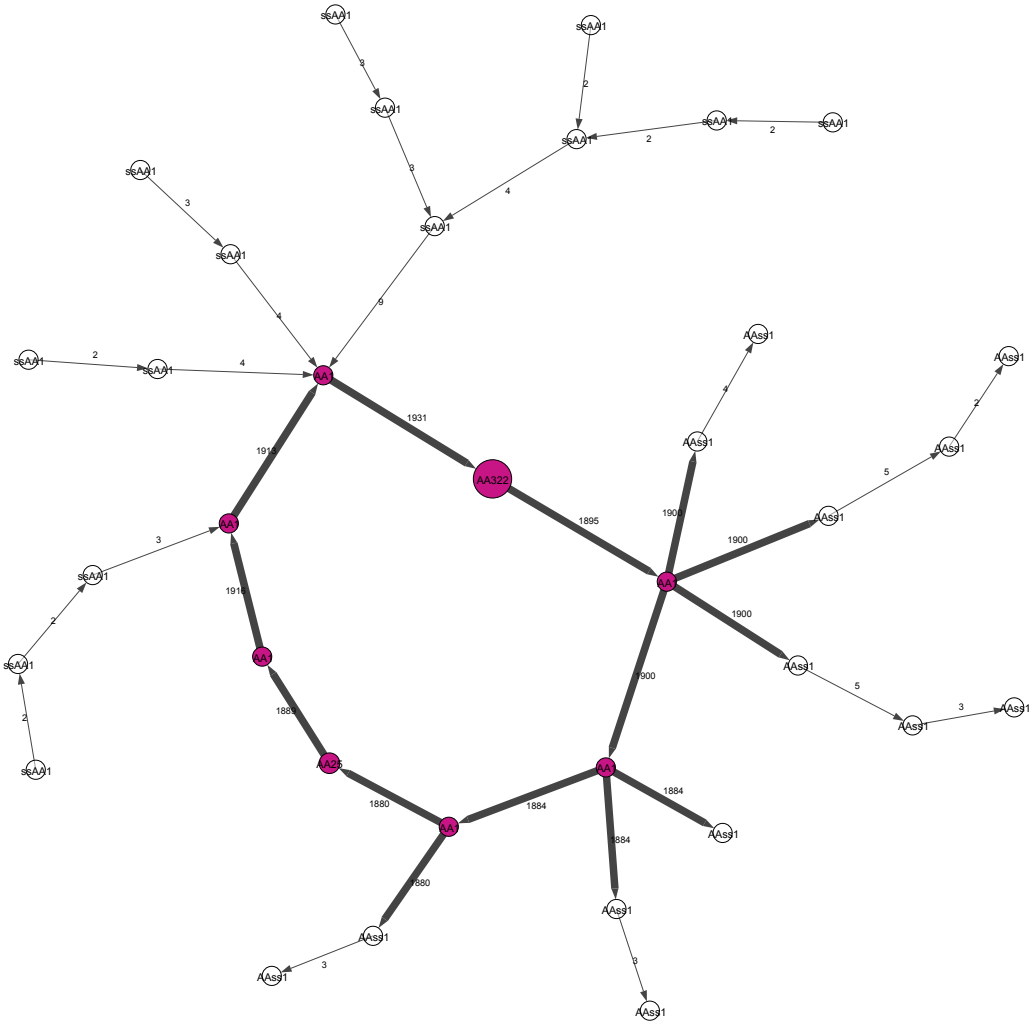


Figure 6.10: Visualisation of one satellite in genome – `TRSA_OS`
with base sequence of 353bp. Thickness of edges and circle structure indicate satellite.

Testing sequence (25Mbp) always contained one repeat, filling scale of repeats was set to 80%.

- **HELITRON7_OS** – Helitron is long 10153bp and has a very unusual structure as it contains micro-satellites. It was inserted 200 times into genome. In a snippet (figure 6.11) of the helitron sequence we can clearly see that is a segment where `at` and `gt` bases are repeated many times. Visualized graph is displayed on figure 6.12 where helitron is distinguished with purple colored node – mentioned micro-satellites segment have extremely high coverage.

- **OSTE22** – is a DNA transposon with length 2037bp. Inner structure is very complicated, is seems that in the middle of repeat there is a short continuous part (around

```
...atcatctgcaatgatattagtaaatgtttattttacaggggataaaagaataaggaaaagaaaagga
gataaacaggagcaagacactgaagctaggaagtacaaaaaggtaaaaaacaataacgagctggaaaatt
ttatatatatatgtgtatatatatatatatatgtgtatatatatatatatatgtgtgtgtctatatatat
atatatgtgtatatatatatatatgtgtatatatatatatatgtgtatatatatatatatgtgtatatat
atatatatatatatatatatatatatatatatatatatatatatatatatatatatatattg
tactggatgtgttgtaccaatgaccaaaaagttatattatgctggatgcgtagacataccaagatcc...
```

Figure 6.11: Snippet of helitron sequence with micro-satellite sequence highlighted in red. Bases `at` and `gt` bases are repeated many times.

200bp) but the most repeat is branched satellite-like structures combined with mutated segments with are present – on visualization (figure 6.13) we can distinguish bubbles which are typical for mutation and segment connect in circle – satellites and their combination. Transposon was inserted 1630 times into testing genome.

- **OSMAR1** – Mariner/Tc1 repeat type with 5265bp, inserted 3850 times into genome. It creates almost mirrored structure that is caused by repeating of smaller sequences inside repeat structure. The visualization is on figure 6.14.

Genome with space between repeats

Genome with space between repeat represents the easiest option for repeats reconstruction. Genome (25Mbp) was generated with spaces of length 20bp and 100 repeats of various types were inserted - each repeats is inserted approximately 100 times. With such number of repeats the graph size rises and visual analysis becomes difficult. There is another problem if repeat - typically is satellite is in the genome, the average of coverage is too high and it is possible that the most of repeats will not be reconstructed. It possible to rise parameter of cuttoff for chimer features removal in order to separate graph into sub graphs – in each graph there is a small amount of repeats that are easily handled by reconstruction phase. But we may lose also important information previously stored in original graph. For testing, sequence without satellites was generated. The results of reconstruction are positive:

- 84% of all repeats found:
  - 80% of found repeats had exact length and sequence
  - 20% were almost correct – there are only mistakes at the beginning or the end of repeat - several (1-30) bases are missing or added to sequence
- Rest of repeats were not found:
  - Repeats have low coverage, thus they are unrecognisable automatically.
  - Special internal structure prevents from finding them.
  - Several of repeat may have been eliminated by chimeric features removal.

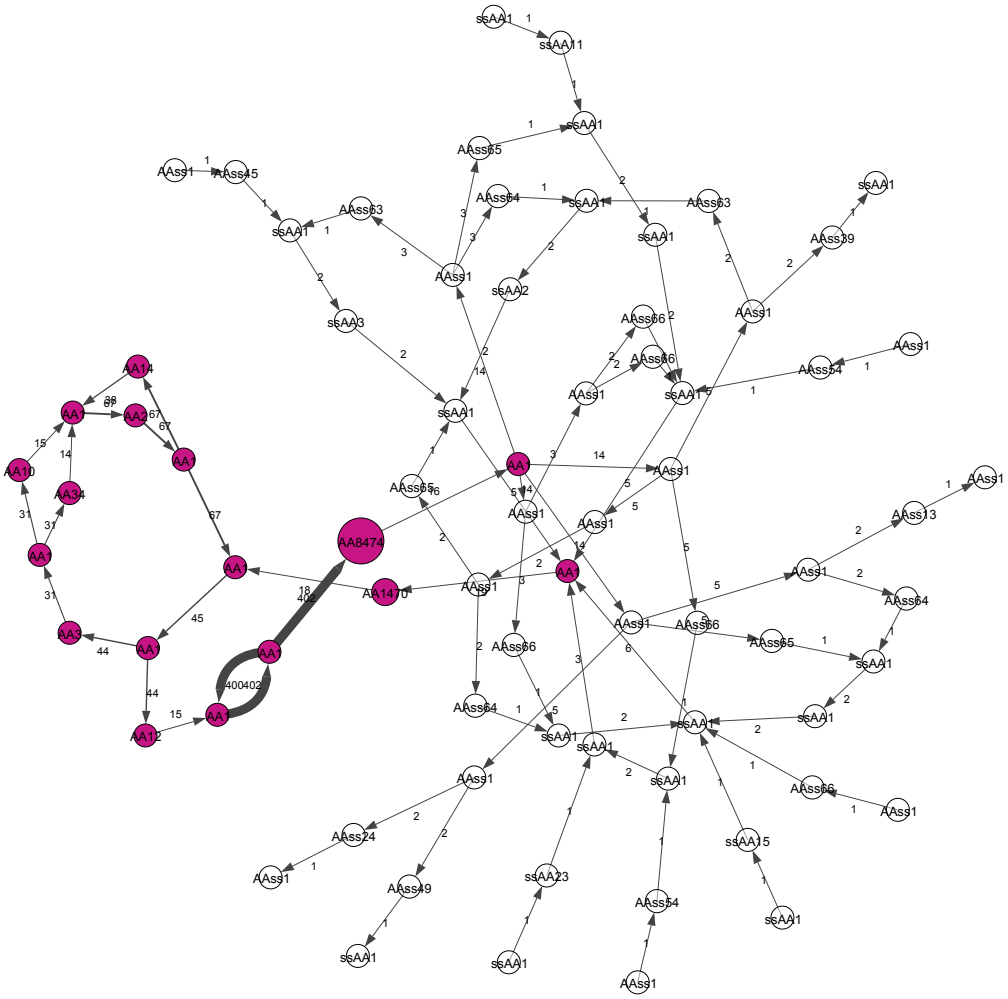


Figure 6.12: Visualisation of **HELITRON7_OS** with micro-satellites. Purple nodes contain only sequence of helitron – micro-satellites are distinguished by extremely thick edges. There is another interesting part on the left of helitron – it may be another satellite with mutated segments.

Genome without space between repeats

In sequence without spaces, there are regions where repeat immediately connects to another repeat or to the same repeat. This situation results into graphs where repeats are closer to each other. Sometimes repeat is connected to itself – it negatively influences reconstruction phase. Longer sequence than original one is considered as whole repeat.

For testing genome (50Mbp) without spaces was generated and 100 repeats were inserted. For testing sequence without satellites was generated. Also known repeat with unusual internal structure were omitted because they misinterpret the reconstruction. The results of reconstruction are as follow:

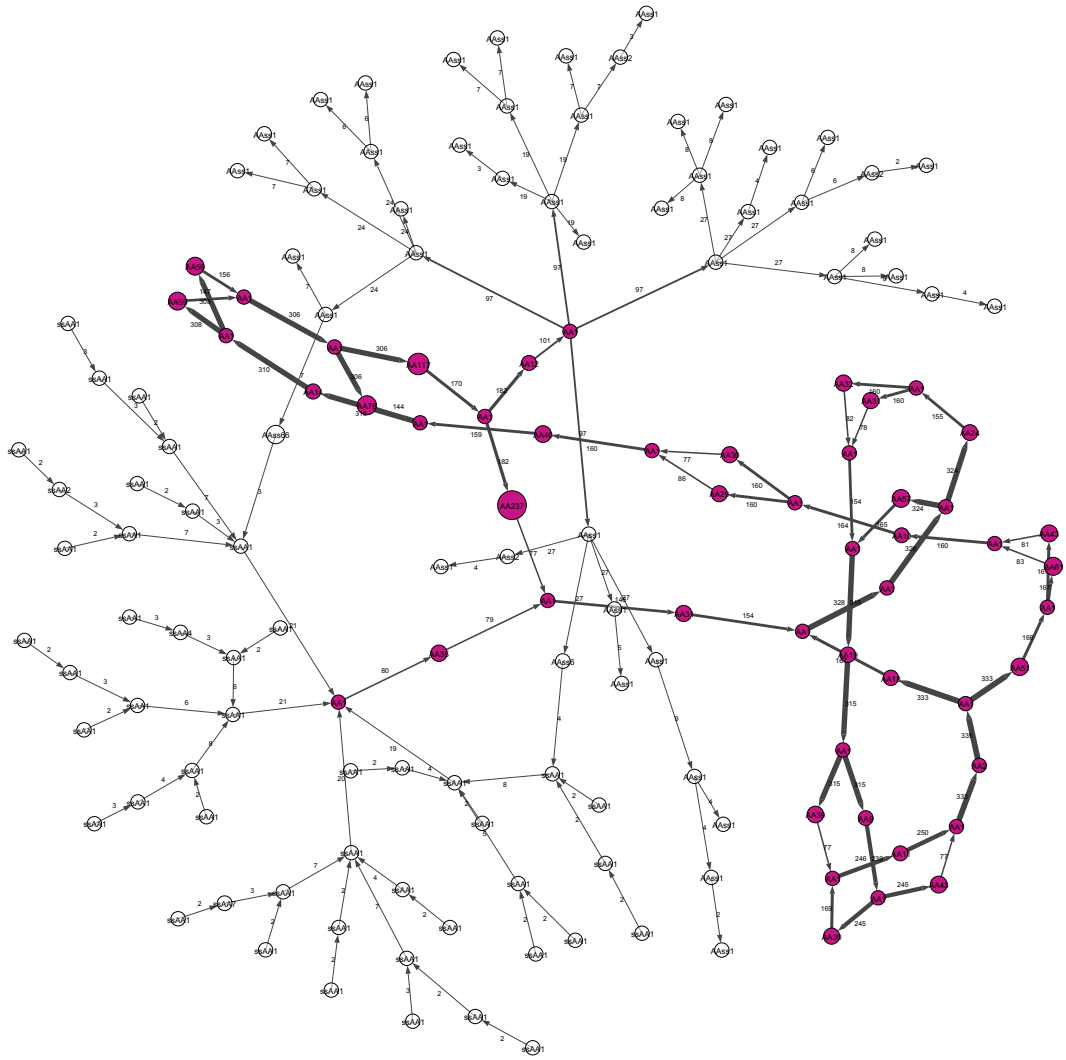


Figure 6.13: Visualisation of **OSTE22** DNA transposon. The structure is very complicated as it combines straight segments with satellites and mutated segments.

- 79% of all repeats found:
  - 60% of found repeats had exact length and sequence
  - 40% were almost correct – there are mistakes at the beginning or the end of repeat - several (1-100) bases are missing or added to sequence because sometime the genome contains two or same repeats in immediate succession. There were also some fragments of repeats reconstructed – but they can be omitted from results because of short length (less than 100bp),
- Rest of repeats were not found:
  - Repeats have low coverage, thus they are unrecognisable automatically.
  - Special internal structure prevents from finding them.

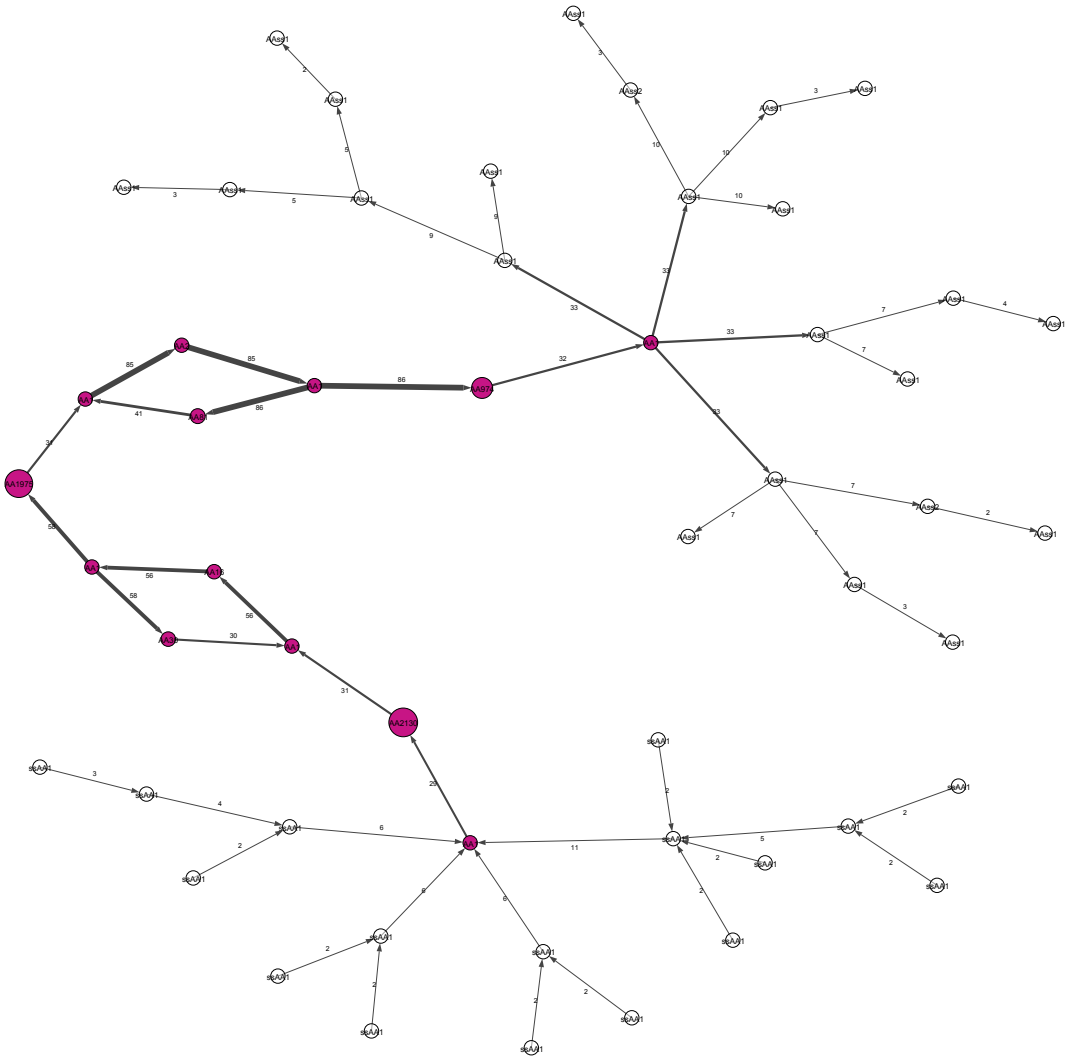


Figure 6.14: Visualisation of **OSMAR1** Mariner/Tc1 repeat. Repeating of smaller sequences inside repeat structure causes bubbles that leads to mirror structure.

– Several of repeat may have been eliminated by chimeric features removal.

– No spaced genome may generate unexpected structures rarely.

Mutation influence

Mutation in repeats is crated according to mutation rate parameter which define how much percent of bases will be mutated. Mutation at the same rate is applied to all inserted repeats. As results we get typical features of graph – tips and bubbles. During graph transformation those features are eliminated. Some bubbles may still exist in graph because of their complexity. During reconstruction phase it may not be possible to found exact sequence. Results of reconstruction of mutated repeats require different approach. For each repeats we get more possible sequences – each of them must be compared to the

original one and results must be aggregated. For genome without spaces it is possible to find sequence with a good accuracy but some of sequences are only fragments of original repeat.

Genome with repeats insertions

Genome with repeats inserted into another repeats is the most complex for analysis. The difficulty arises from repeats fragmentation. We can expect that only small amount of repeats will not be fragmented. Due to such circumstances it is expected to reconstruct only fragments of repeats sequence – but even we will no acquire full sequences, results could be used for further analysis.

Long repeats influence

Situation is more complicated, if we insert repeats with very long sequence into genome. Especially when genome consist of very long repeats (more than 10Kbp) – insertion of small repeats into long ones will result into total destroying of original long repeat and reconstruction of such sequence may not be possible at all.

For testing purposes, genome (50Mbp) with insertions was generated. Inserted repeats (100) do not consist of repeat with special structure. Long repeats with more than 5000bp were also omitted. Threshold for repeats reconstruction was lowered to the half of average of coverage. The graph is split into many sub graphs because of chimeric connection elimination. The reconstruction results are as follow:

- 80% of all repeats found:
  - 0% of found repeats had exact length and sequence – only fragments were found.
  - The rest of repeats: there are only fragments of all repeats. Position of fragment in repeat sequence differs - it may at the beginning, in the middle, or at the end. Average length of found fragmeny was around 25% of original sequence length. Even this results are not much positive, further deep analysis of fragments could be performed.

Repeats of the same families

It is a very common situation that many repeats from one family are present in the genome. The core sequence of those repeats is very similar and the major differences are expected at beginning and at the end of repeat. If we try to visualize repeat of the same family, similarities in sequences cause integration of the repeat into each other. Although sequences of each repeats can be reconstructed, in our test case it is expected that some integrated sequence of both repeats is reconstructed. Visualization is displayed on figure 6.15.

Described variation was tested on two repeats of *Gypsy* family:

- **RIRE3A_LTR** – 3167bp, Gypsy LTR retrotransposon.
- **RIRE3_LTR** – 3151bp, Gypsy LTR retrotransposon.

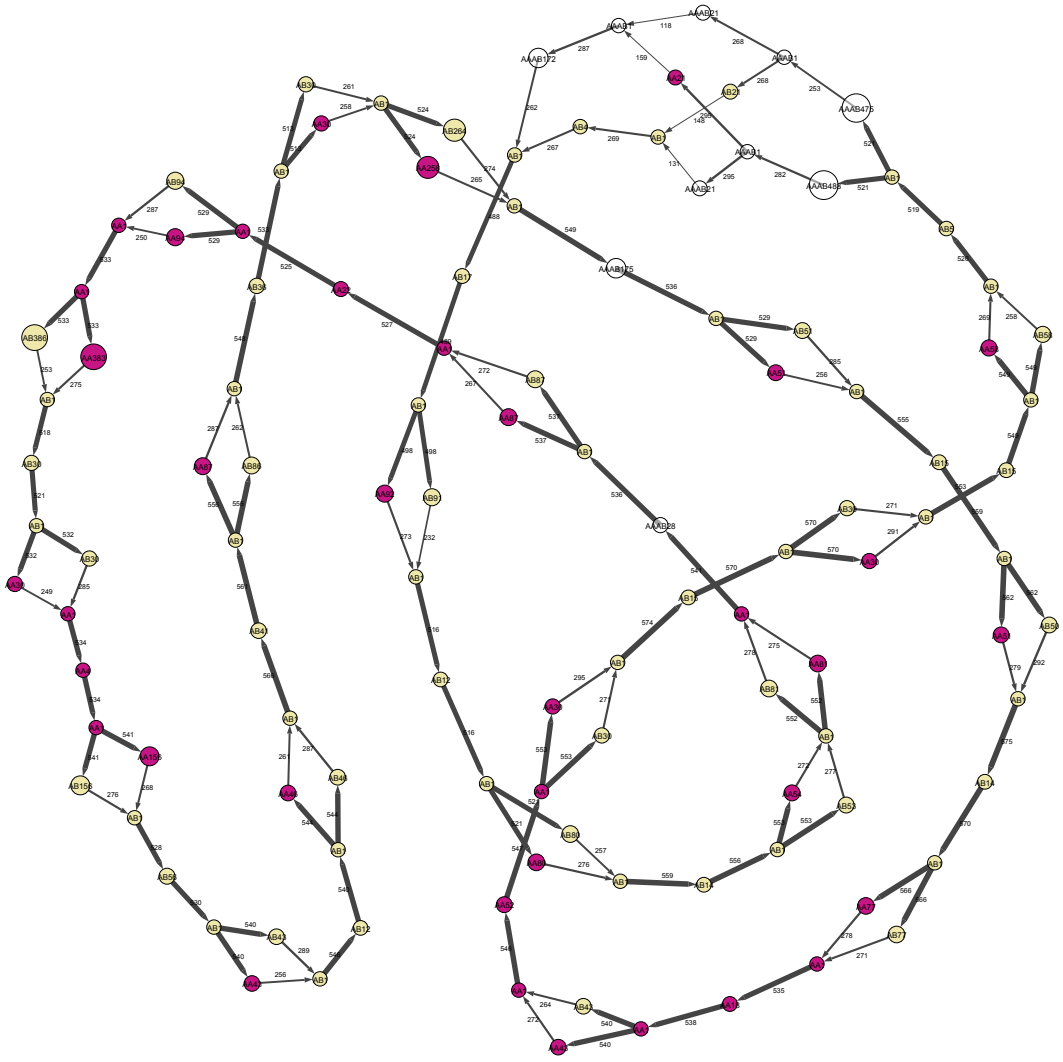


Figure 6.15: Visualisation of **RIRE3A_LTR** (purple) and **RIRE3_LTR** (yellow) retrotransposons of Gypsy family. Transposons are integrated to each other and create bubbles – there are mutations in the sequence. In some segments one repeat merges into next repeat and vice-versa.

Conclusion

In the diploma thesis different types of repetitive segments of DNA were described. Existing tools for reconstruction of tranposable elements were summarized and chosen method for genome representation was defined – de Bruijn graph. That method proved to be easily implemented and very efficient in the view of time and space complexity. Useful methods for graph transformations were implemented – consecutive segments merge, tips, bubbles and chimeric connections removal. Transformations and errors correction are required either for visualization, or for repeats reconstruction. Transformed graph is quickly visualized and can be visually analyzed. That helped me a lot with testing and evaluation. Finally I was able to determine which segments of graph belong to repeats and which belong to random sequence. Final part of the thesis described testing methods and evaluated results of graph transformations and repeats reconstruction. Test scenarios for graph transformations shown that great reduction of edges and nodes count can be done. The most interesting part of testing were test cases for reconstruction of repetitive segments. It turned out that the reconstruction is very challenging and requires sophisticated approach. Several types of genome were tested – from the least complex to the extreme complex. Reconstruction works correctly on simpler genome. There are many types of repeats that have special internal structure, so they are very difficult to be reconstructed. Complex variants of genome where repeat insertions into another repeat is allowed and repeats are inserted with mutation almost make reconstruction very difficult, perhaps impossible, at best only fragments can be reconstructed.

Possible future improvements

There are possible improvements that could lead to more accurate repetitive segments reconstruction:

- **Opposite strand and pair reads usage** – they may give us better knowledge of the genome structure. Therefore, algorithms could altered to analyze the graph in a way which could result into higher accuracy of repeats reconstruction, possibly it might help with visual analysis.
- **Sophisticated heuristics for reconstruction of fragmented sequences** – now the program uses quite simple but efficient method for repeats reconstruction. Perhaps adaptive version could designed that will adapt according to current properties of graph and find longer repeats fragments in the complex genomes.
- **Interactive exploration of graph** – such interactive exploration may help with better understanding of found structures and therefore for tool adjustment in order to be more accurate.

Bibliography

[1] Cairo graphics. https://www.cairographics.org/, 2014.

[2] Python-igraph [online]. http://igraph.org/python/, 2015. cit. 2016-01-05.

[3] Biopython. http://biopython.org/wiki/Biopython, 2016.

[4] Bitbucket. https://bitbucket.org/, 2016.

[5] Fastq format. http://maq.sourceforge.net/fastq.shtml, 2016.

[6] Git. https://git-scm.com/, 2016.

[7] Illumina sequencing. http://www.illumina.com/technology/next-generation-sequencing/solexa-technology.html,
2016.

[8] Pycharm. https://www.jetbrains.com/pycharm/, 2016.

[9] Python 2.7.11 documentation [online]. https://docs.python.org/2/, 2016. cit. 2016-01-05.

[10] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, 2008.

[11] C. Feschotte and S. R. Wessler. Treasures in the attic. *Proceedings of the National Academy of Sciences of the United States of America*, 98(16):8923, 2001. ISSN 00278424.

[12] Manuel A. Garrido-Ramos. Satellite dna in plants. *Cytogenetic and Genome Research*, 146(2):153–170, 2015.

[13] J. Jurka, V.V. Kapitonov, A. Pavlicek, P. Klonowski, O. Kohany, and J. Walichiewicz. Repbase update, a database of eukaryotic repetitive elements. *Cytogenetic and Genome Research*, 110(1–4):462–467, 2005.

[14] Eduard Kejnovsky, Jennifer S. Hawkins, and Cédric Feschotte. Plant transposable elements. *Plant Genome Diversity Volume 1*, (1):17, 2012. ISBN 978-3-7091-1129-1.

[15] Margaret G. Kidwell. Transposable elements and the evolution of genome size in eukaryotes. *Genetica*, vol. 115(issue 1):49–63, 2002. ISSN 00166707.

[16] Donald Ervin Knuth. *The art of computer programming / Vol. 3, Sorting and serching*. Addison-Wesley, Upper Saddle River, 3rd ed. edition, 1998.

[17] Ben Langmead. De bruijn graph assembly [online]. http://www.cs.jhu.edu/langmea/resources/lecturenotes/assemblydbg.pdf, 2015. cit. 2016-01-05.

[18] Martin Munoz-Lopez and Jose Garcia-Perez. Dna transposons. *Current Genomics*, vol. 11(issue 2):115–128, 2010-04-01. ISSN 13892029.

[19] Petr Novák, Pavel Neumann, Jiří Pech, Jaroslav Steinhaisl, and Jiří Macas. Repeatexplorer. *Bioinformatics*, 29(6):792–793, 2013020315.

[20] Janka Puterová. Porovnání eukaryotních genomů, 2015.

[21] Thomas Schmidt. Lines, sines and repetitive dna. *Plant Molecular Biology*, 40(6):903–910, 199908. ISSN 01674412.

[22] S. Evan Staton and John M. Burke. Transposome. *Bioinformatics*, 31(11):1827–1829, 2015020601.

[23] Susan R. Wessler. Transposable elements and the evolution of eukaryotic genomes. *Proceedings of the National Academy of Sciences of the United States of America*, 103(47):17600–17601, 2006. ISSN 00278424.

[24] T. Wicker and F. Sabot. A unified classification system for eukaryotic transposable elements. *Nature Reviews Genetics*, 8(12):973, 2007. ISSN 14710056.

[25] Daniel Zerbino. *Genome assembly and comparison using de Bruijn graphs*. Dissertation thesis, University of Cambridge, Cambridge, United Kingdom, 2009.

[26] Matthias Zytnicki, Eduard Akhunov, and Hadi Quesneville. Tedna. *Bioinformatics*, 30(18):2656–2658, 2014060915.

Appendices

List of Appendices

Appendix A

DVD content

- **/src** – source code of the program.
- **/install** – installation instructions.
- **/tex** – source code of thesis written in LaTeX.
- **/man** – user manual for program configuration and execution.
- **/xbikar00.pdf** – electronic version of the thesis.